(19) European Patent Office
Office européen des brevets

(11) Publication number : **0 528 631 A2**

(12) EUROPEAN PATENT APPLICATION

(21) Application number : **92307377.9**

(51) Int. Cl.[5] : **G06F 15/72**

(22) Date of filing : **12.08.92**

(30) Priority : **13.08.91 US 744630**

(43) Date of publication of application :
24.02.93 Bulletin 93/08

(84) Designated Contracting States :
DE FR GB

(71) Applicant : **XEROX CORPORATION**
Xerox Square
Rochester New York 14644 (US)

(72) Inventor : **Brotsky, Daniel Carl**
1162 Colusa Avenue
Berkeley, California 94707 (US)
Inventor : **Rabin, Daniel Eli**
675 Orange Street
3 New Haven, Connecticut 06511 (US)
Inventor : **Levy, David Myron**
3412 Cowper Street
Palo Alto, California 94306 (US)

(74) Representative : **Goode, Ian Roy et al**
Rank Xerox Patent Department Albion House
55 New Oxford Street
London WC1A 1BS (GB)

(54) Electronic image generation.

(57) A graphics editor generates an appearance construction graph (ACG) which represents an image formation process as a directed acyclic graph (DAG). The ACG includes nodes representing image fragments (sources), image transformation operations (transforms), and outputs (viewable images - sinks) which can be displayed, for example, in a window of a display screen, and manipulated to construct and edit complex images. Accordingly, the displayed ACG is a user interface that permits complex graphics images to be constructed in a straightforward manner, and thus easily understood and manipulated (edited) by an operator. The nodes in the ACG are selectively interconnected by links (edges) therebetween. The complex image represented by the ACG can be constructed and displayed in a viewer window on the display screen by "running" the ACG. The graphics editor "runs" the ACG by performing operations on image fragments as defined by the interconnected nodes of the ACG.

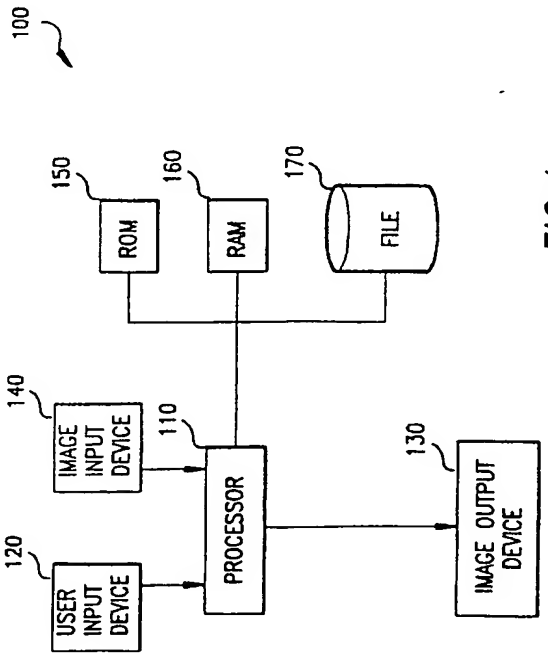


FIG. 1

EP 0 528 631 A2

The present invention relates to graphics editors, and more particularly to graphics editors for constructing and editing complex graphics images, and to methods for forming and editing complex graphics images through the use of a user interface.

The standard What-You-See-Is-What-You-Get (WYSIWYG) point-and-click editing paradigm presents difficulties for the construction of complex graphics images. If the user can view only the final WYSIWYG image, it is hard to see and understand its composition, that is, its components and their relations. Consequently, it is difficult to point to the components and modify them. It is also difficult to reuse the same image, whether transformed or modified, without interactively copying it one or more times.

For example, to operate on a graphics image, users need to indicate - that is, refer to - the part or parts of the image they wish to affect. This is often done by pointing with a mouse. Experienced users know that when manipulating complex images, it can be difficult to find and point to the relevant parts, especially when there are dense clusters of overlapping figures. Related to this, users sometimes find that parts of the image they can see are not recognized by the editor, so cannot be pointed at and operated on independently. A simple example is when a rectangle is created using a "rectangle" tool. It typically is not possible to select one of the sides of the rectangle and move it away. To do this, the rectangle must be created as four line segments. There is no ability to reparse or reregister the rectangle - to decompose it into parts other than the ones out of which it was first constructed. This ability to reregister is closely related to the problem of reference, since, the registration of an image determines the parts that can be referred to.

Another problem which arises with the WYSIWYG display screen is the inability of an operator to determine the graphical domain in which a portion of a complex image is located. A number of commercially available text/graphics editors are capable of constructing images in both the bitmap and vector domains. For example, Superpaint® is a graphics editor which incorporates some painting (bitmap) and drawing (vector) capabilities. However, unless the user of the system knows (based on experience) in which domain a particular portion of an image is created, the user may attempt to perform editing operations on that image portion while in a domain in which that editing cannot be performed.

Accordingly, a need exists for a graphics editor which conveys the underlying components and operations which are performed to produce a complex image to the operator, and which permits the operator to construct and edit complex images easily.

US-A-4,901,063 discloses an image processing apparatus which displays both a sample image as it appears before a selected image process is executed, and a processed sample image corresponding to an image resulting from the selected image process. With the apparatus, an operator can obtain processed sample images and can perform image processing on an original image by inputting a processing command.

US-A-4,611,306 discloses a word processing system having a display menu/chart key for toggling back and forth between a graphics image and a menu display used to define a graphics image. The system provides an interface which allows an operator to check and revise the definition of a graphics image visually. By using the menu/chart key, the operator can see how changing a menu item will cause a graphics image to be changed. When the key is pressed while a graphic image is being displayed, a most recently processed menu is again displayed. This editing system does not permit the viewing of an entire complex image formation process, or the simultaneous viewing of an image formation process and the image formed thereby.

US-A-4,928,247 discloses a host system which executes one or more application programs which result in graphics data structures. These graphics data structures are then continuously and asynchronously traversed. In particular, hierarchical data structures are built in memory to permit the definition of complex data structures representative of the primitives, attributes and geometric transformations of three dimensional objects. This data structure is then asynchronously traversed by a structure walker.

US-A-4,021,976 discloses a method and system for generating dynamic interactive visual representations of complex information structures within a computer. Boundaries of the information system are established, and a set of mathematical relationships is provided which indicates a degree of correlation between parameters of interest to a user and segments of information contained within the boundaries. A visual display is generated for the user which has a plurality of iconic representations and features corresponding to the parameters defined by the mathematical relationships. The iconic representations and visual features of the visual display change with movement of the mathematical relationships within the boundaries of the information system according to the degree of correlation between parameters of interest and a segment of information through which the mathematical relationships are passing. The shapes of the icons can be used to denote the type of data represented by the icons.

US-A-4,860,204 discloses a method for constructing computer programs through use of visual graphical representations. Computer programs are illustrated as visual roadmaps of an intended sequence of actions. Each operational entity of a program graph displayed on a screen is represented as an elemental unit. The elemental unit is a multidimensional graph-

ical unit of programming information which has four modes, or "layers" of operation. The elemental units reside in very general form in a work station library and are optimized by specialization. The system permits programs to be written using icons instead of by writing source code.

US-A-4,953,106 discloses a computer implemented technique for drawing directed graphs providing reduced crossing and improved picture quality.

It is an object of the present invention to provide a graphics editor which permits complex images to be formed and edited easily, and in a straightforward manner.

In accordance with the present invitation, a graphics editor generates an appearance construction graph (ACG) which represents a complex image formation process as a directed acyclic graph (DAG). The ACG includes nodes representing image fragments (sources), image transformation operations (transforms), and outputs (viewable images - sinks) which can be displayed, for example, in a window of a display screen, and manipulated by a user to construct and edit complex images. Accordingly, the displayed ACG is a user interface that permits complex graphics images to be constructed in a straightforward manner, and thus easily understood and manipulated (edited) by an operator. The nodes in the ACG are selectively interconnected by user defined links (edges) therebetween. The complex image represented by the ACG can be constructed and displayed in a viewer window on the display screen by "running" the ACG. The graphics editor "runs" the ACG by performing operations on image fragments as defined by the interconnected nodes of the ACG.

An operator constructs an ACG graphically in a graph view window by selecting node types, placing icons for these types in the graph view window, and creating links (edges) between the nodes. The operator can add or delete nodes or links at any time. Additionally, the image fragments (source node contents) can be edited or imported by, for example, reference to a library of source images. When editing image fragments, another window (i.e., an editor window) is displayed which permits the operator to edit the contents of a selected source node. Once an ACG has been constructed, and content created or imported for each source node, the ACG can be run and its results displayed.

The present invention is implemented by directly representing ACGs in an object-oriented graph representation language. The nodes in the graph are objects which keep track of their connections to neighbors and which respond to a standard set of graph traversal messages. The graphics functionality of nodes is implemented by making their objects inherit functionality not only from the node's class (type of node) but also from a graphics operator class which responds to standard operation messages such as reading inputs and providing outputs. Node types correspond to subclasses of the generic operator class which implement functionality appropriate to the node type.

As an operator uses the graph-construction interface of the present invention to build an ACG, the editor builds the ACG in the underlying object model. When the operator requests that the ACG be executed, the editor runs a graph traversal algorithm which, as it walks the graph (e.g., from leaf nodes toward root nodes), invokes the graphics operations associated with each source, transform, and sink. This has the effect of constructing and displaying the appropriate appearance (complex output image). That is, the editor executes ACGs by propagating graphics information along the dataflow arcs which connect the nodes of the ACGs.

Thus, the present invention permits an operator to control complex image formation by manipulating a displayed graph of the image formation process. An operator is able to develop an image formation process by forming and modifying a graphical display of the image formation process.

The invention will be described by way of example with reference to the following drawings in which like reference numerals refer to like elements and wherein:

Figure 1 is a block diagram showing general components that can be used with the present invention;

Figure 2 is a view of a display screen having an ACG in one window and a constructed output image in another window;

Figure 3 is a view of a display screen having windows containing different types of source image editors;

Figure 4 is a view of a display screen having an ACG of Figure 2 modified and the Viewer windows which result;

Figure 5 is a view of a display screen resulting from a further modification of the Figure 4 ACG;

Figure 6 is a view of a display screen resulting from a modification of the Figure 5 ACG;

Figure 7 is a view of a display screen illustrating the modification of a polygon defined by Euclidean line segments;

Figure 8 is a view of a display screen containing a second ACG and Viewer windows as well as a Prompt Window;

Figure 9 is a list of node icons;

Figures 10A and 10B are ACGs of source nodes attached to sink nodes;

Figure 11 is an ACG of a Euclidean line segment source node attached to a Viewer node by a scan-conversion transform;

Figure 12A is an ACG of a parameterized scan-conversion node attached to a Viewer node;

Figure 12B is an ACG of a clipping transform at-

tached to a Viewer node;

Figure 13 is an enlarged view of the Figure 8 ACG;

Figure 14 is a block diagram of the modules forming the graphics editor of the present invention;

Figure 15 is a flow diagram of an image formation process according to the present invention;

Figure 16 is a flow diagram of the ACG construction process of the present invention;

Figure 17 is a flow diagram of the ACG execution process of the present invention; and

Figures 18A and 18B are flow diagrams of ACG editing processes.

The present invention relates a graphics editor, which produces a desired graphics image (or appearance). The edition manipulates image data drawn from more than one figural model (graphical domain). A figural model as defined herein is what is referred to in graphics literature as an imaging model - a specification of constructable appearances in terms of the operations available for marking a surface. Specifically, the present invention can manipulate both raster graphics apparatus of (bitmap graphics), which are figures defined over a discrete, cellular grid, and line-segment graphics (vector graphics) which are figures constructed using line segments in the real plane (where real numbers are approximated by computer floating point numbers). The apparatus can also be extended to include other figural models, such as, for example, text, video and three- dimensional figural models.

In the apparatus of the present invention, graphics images are constructed by directly manipulating underlying image fragments and by applying graphics operators (transforms) to one or more underlying image fragments in order to produce new images. Specifically, the apparatus permits the application of arbitrarily defined transformations to images, including transformations which translate images from one figural model to another so that the "transformed" image can be further operated on in the new figural model.

The user interface provided by the apparatus explicitly represents all the underlying image fragments being used, all the operations being applied to them, and all the places where output images should be produced. Specifically, an operator works by manipulating a graphical flow diagram whose nodes represent image fragments (sources), operations (transforms), and outputs (sinks). This allows the reuse of image fragments and transformed image fragments (the outputs of transform nodes) without confusion as to what effect changes in the sources and operations will have on the final output image.

The modular construction of the apparatus means that it is completely extensible, both with respect to the transforms available and the underlying figural models from which one may draw source images. Specifically, new transducers can be integrated which perform arbitrary image transformations or which implement new figural models. In order to integrate a new figural model, all that is required is the integration of image sources from that new figural model and operations (transforms) for converting an image in the new figural model to one of the existing figural models. Once an image from a new figural model is converted to an existing figural model, operations appropriate for the existing figural models can be performed on the converted image.

The present invention is implemented on a conventional hardware system 100 which includes a microprocesser 110 for receiving signals from, and outputting signals to, various other components of system 100, according to one or more software programs run on microprocessor 110. A user input device 120 such as, for example, a mouse, a keyboard, a touch screen and/or combinations thereof, is provided for permitting an operator to interface with the editor by, for example, supplying control signals thereto. An image output device 130, such as, for example, a video display monitor, or a printer, can be provided for outputting image data. An image input device 140, such as, for example, a scanner, can be provided for scanning images which are then stored in memory as image sources. Image sources can also be supplied from, for example, discs.

Microprocessor 110 includes a read only memory (ROM) 150 for storing operating programs and image data, used to construct and edit images. A random access memory (RAM) 160 is provided for running the various operating programs, and additional files 170 can be provided for RAM overflow.

One embodiment of the present invention was implemented in version 1.2.2 of Macintosh Allegro Common Lisp (Macintosh is a registered trademark of Apple Computers; Allegro Common Lisp is a trademark of Franz, Inc.), a dialect of Common Lisp available from Apple Computer. This prototype will run on any computer which supports Macintosh Allegro Common Lisp, however, a Macintosh II computer (or faster) with at least five megabytes of RAM and a 483mm diagonal monitor is preferred so as to provide an editor which is adequately fast.

In order to convey the manner in which the apparatus of the present invention works, and the basic mechanisms which provide it with its novel capabilities, examples of the user interface will now be provided.

The apparatus displays an ACG which represents the image manipulation processes which are performed on one or more image fragments to produce one or more final output images. The ACG is displayed in, for example, a window on a monitor's display screen. When executed, the final output images created by the ACG (some of the output images may be components or intermediate images of other out-

put images) are also displayed on the monitor display screen in their own respective windows. Thus, the present invention provides a type of two-view graphics editor. One view shows the final (i.e., the WYSIWYG) image(s), and the other view displays the ACG representing the construction of that (those) image(s) (i.e., the image fragments and operations performed on the image fragments which result in the output image(s)).

Example One

Figure 2 shows a simple, first example of a display screen which includes an ACG displayed in the lefthand window and the final output image displayed in the righthand window. The final image is a triangular portion of an image of Einstein superimposed on a dotted background. The ACG has three types of nodes: source nodes; transform (operator) nodes; and sink nodes.

Source nodes, which are the bottom-most ("leaf") nodes in the ACG, represent image fragments or other source data. In the ACG of Figure 2, the source nodes are labelled (from left-to-right) C,B,F,D. The icon for each of these nodes indicates the type of source data it contains. (Figure 9 is a list of the various icons which can be selected). Thus, in Figure 2, the node labelled C contains a uniform raster pattern (a "background" raster image), while node B contains a set of vectors, or line segments. The actual contents of each of the four source nodes are illustrated in the display of Figure 3. Each node's contents is displayed in a separate window which is labelled with the same letter as that node. For example, the window entitled "Segment Set B" corresponds to the source node "B" in the ACG. Window B thus reveals that the content of node B includes three line segments intersecting to form a triangle. When displayed in a window on the monitor screen, the contents of these nodes can be edited by an operator inputting commands. For example, the operator can manipulate a cursor and/or press buttons (on a mouse or keyboard) to edit the contents of a source node. Additionally, source images can be imported from external libraries in a conventional manner. The editing steps can be performed in conventional ways using conventional software.

Transforms (operator nodes) are the intermediate nodes in an ACG. In Figure 2, the three icons labelled T1, T2, and T3 are transforms. Transforms operate on image fragments, i.e., they take image fragments and other data as input and produce a new image fragment as output. Inputs to transforms come either from source nodes (primitive image fragments) or from the outputs of other transforms (which can produce intermediate images). Thus, the transform T1 has two inputs, both of which are the content of source nodes B and F, while T2 has two inputs, one the content of source node D, the other the output of transform T1. (Source nodes can be thought of as transforms that have no inputs and produce their content as output). The inputs and outputs of transforms are typed. Thus, T3 takes two or more raster images as input and produces as output a raster image constructed by superimposing the input images one on top of the other.

Sink nodes are the root nodes in an ACG, indicating where an image is to be output. Sink nodes, such as node A in Figure 2, can represent windows or "viewers" on the display screen. In the present embodiment, inputs to sink nodes must be raster images. Sink nodes might also represent other output devices such as printers or files, and might be capable of accepting types of input other than raster. For example, monitors could accept images in the video figural model, and a plotter could accept images in the vector figural model. The window labelled "Viewer A" in Figure 2 corresponds to sink node A in the Figure 2 ACG.

Images are synthesized (or created) by traversing the image fragments and operations represented by the ACG with a structure walker. In the illustrative embodiment, image fragments flow upward (from leaf nodes to root nodes), are transformed and combined by transforms, and are displayed in, for example, windows on the display screen.

Referring to Figure 2, the final output image, displayed in viewer A, is the output of transform T3. T3 is the superposition transform, it superimposes the output of transform T2 on top of the background raster (the grid of dots) represented by source node C. Transform T2 is a clipping operator which clips out a polygonal section of a raster image. One input of transform T2 is the raster image to be clipped (source node D, whose contents is a raster image of Einstein), the other input is a set of vectors specifying the polygonal mask (the output of transform T1). Transform T1 is a simple "recognition" operator: a "polygon finder". Given a set of vectors (the content of source node B) and a point (the content of source node F), transform T1 finds and outputs the smallest polygon enclosing that point.

Images are created in three stages:

1. ACG creation and modification. The operator creates an ACG graphically by inputting commands for selecting node types (from a displayed menu or list of node types), placing icons for these node types in the ACG window, and creating links (edges) between the nodes. The user can add or delete nodes or links at any time.

2. Image fragment (source node content) importation and editing. A different type of editor is provided for each type of source node, so that the contents of that source node can be edited. Each "source node editor" includes a window labelled with the same letter as that source node's icon in the ACG, in which the contents can be edited.

With reference to Figure 3, the window labelled "Segment Set B" permits the operator to move and create line segments which are used to define a polygon. The window labelled "Point F" permits the location of a point, used by "polygon finder" T1 to be changed. The window labelled "Raster Constant D" permits an operator to define a raster image fragment by selecting from a menu of background patterns. Selections are made in a conventional manner by, for example, moving a cursor and striking keys (on a mouse and/or a keyboard). Additionally, source node contents can be imported from libraries such as, for example, discs.

3. Image Viewing. Once an ACG has been constructed and content created for each of its source nodes, the ACG can be executed and its results displayed.

The ACG or source node contents can be modified at any time by providing the appropriate control signals to the microprocessor using, for example, the cursor and key strokes. Figure 4 shows a modified ACG and the resulting images. The triangular clipping of Einstein now has a black border around it, and there is a new viewer window, Viewer E, showing three vectors forming a triangle. The Figure 4 ACG differs from the Figure 2 ACG in the addition of two new nodes -a transform labelled T4 and a viewer node labelled E- and three new edges connecting T3 and T4, T4 and T1, and T4 and E, respectively.

The new transform, T4, is a scan conversion operator: it takes a set of vectors as input and yields a raster image by scan converting each of the vectors. Thus, the border around the clipped Einstein image is the result of scan converting the recognized polygon (which was also used as a mask to the clipping operator T2) and superimposing it on top of the clipped Einstein and the dotted background. The output of transform T4 is also displayed in Viewer E. A comparison of Viewer E with the contents of the window labelled "Segment Set B" in Figure 3 shows that transform T1 has "sliced and diced" the original vectors in node B, outputting only those fragments that form a polygon (in this case, a triangle).

Viewer A in Figure 4 now displays a window, i.e, a picture of Einstein viewed through a polygon (shaped window) displayed against a textured background. In Figures 5 and 6, the window metaphor is further extended. These extensions make use of two additional nodes of the same type: a translation transform (T6 and T7 in Figures 5 and 6) translates or displaces an image on the display surface. The inputs for the translation transform are a raster image to be displaced and an amount (delta x, delta y) of displacement. The displacement nodes are source nodes G and H.

In Figure 5, the superposition of Einstein's border and background has been regrouped so a translation operator can be attached to the superposition of Einstein and border. As a result, the displacement node G acts as a "window mover". By changing the x and y values in displacement G (in the editor window for the content of node G) and re-executing the ACG, Einstein plus border can be displaced (moved) by delta x and delta y.

In Figure 6, the same basic move, the addition of translation and displacement nodes in the appropriate places creates yet another effect, i.e., displacement node H acts as a scroll bar. Entering x and y values in displacement H (attached to transform T7) and re-executing the ACG causes just the Einstein raster image to be displaced (scrolled) by delta x and delta y.

Figure 7 illustrates how the shape of the window can be changed by simply editing the vectors from which the window was originally formed and rerunning the ACG. The structure of the ACG in Figure 7 covered by the Segment Set B Window is the same as the Figure 6 ACG.

Example Two

Figure 8 illustrates what an editor screen might look like when an operator has finished constructing a more complex appearance. As in the example of Figures 2-7, the center piece of the user interface provided by the present invention is the large graph view window 10 which is where an operator pieces together the desired appearance by combining image fragments and transforms in a desired way.

The icons used to depict nodes in the ACG are based on the type of the node, rather than its contents. For instance, two nodes containing raster image fragments will be depicted with identical icons even though the image fragments they contain might be different (although they will be labeled with a different letter). Similarly, the nodes for two different applications of a masking operation will be depicted with identical icons because they are both masks. It is important, when viewing an ACG not to confuse the appearance of the icon for a node (which indicates its type) with the appearance output by the node.

Figure 9 shows some different icons which can be used and the types of nodes they represent. Icon 20a is a viewer node. Icon 20b is a background node. Icon 20c is a raster appearance. Icon 20d is a line segment appearance. Icon 20e is a displacement icon. Icon 20f is a point icon. Icon 20g is a superposition icon. Icon 20h is a translation icon. Icon 20i is a scan conversion and parameterized scan conversion icon. Icon 20j is a polygon finder icon. Icon 20k is a mask icon.

Note that square nodes which denote input source images and output constructed images, are labeled with single letters from "A" to "Z". These labels are used for identification. Whenever an operator requests, by inputting a command, such as with the

operation of a cursor and the stroking of keys, via a mouse and/or a keyboard, the graphics editor of the present invention will display a special window which shows the contents of any square node. This window's title will contain that node's letter. In the case of input source images, these windows are editors which can be used to alter the node's image data content. In the case of output images, these windows are viewers only, as the image is completely determined by the ACG itself. For this reason, output nodes are referred to as "viewer" nodes. Figure 8 contains a number of viewer and editor windows to the left of the graph view window 10.

The present invention can be used in a conventional way, i.e., by mousing in windows, and by invoking commands from menus. Additionally, a prompt window 12 can be provided to give feedback on the course of operations invoked.

In order to provide a better understanding of the manner of which the ACG 10 of Figure 8 produces the contents of the viewers in Figure 8, more simplified ACG's will first be described. Figures 10A and 10B are the simplest possible ACG's, since each simply connects an image fragment to a viewer. The ACG of Figure 10A is a viewer connected directly to a "background", i.e., an image fragment containing a raster pattern which is repeated horizontally and vertically to fill its allotted space in the final appearance. The ACG of Figure 10B is a viewer connected directly to a raster appearance, i.e., an image fragment containing a previously-prepared bitmap. Raster appearances can comprise previously scanned images which are stored in a file (library). Figures 10A and 10B illustrate that the present invention can be utilized to output the contents of image sources directly, without any transformation operations being performed thereon.

To construct ACG's in the graph view window, commands are issued (e.g., with mouse clicks) to indicate where nodes are desired. Similarly, a command can be issued to indicate that two nodes should be connected. This can be accomplished, for example, by dragging a cursor between two nodes while pressing a button on a mouse and a key on the keyboard to form an edge between the two nodes. When two nodes are connected by an edge, the one on top is considered the parent and the one below the child. Parent nodes work by taking the outputs of their children as inputs and then producing outputs themselves. Thus, each of the ACG's in Figs. 10A and 10B would require simply two clicks and a drag to construct. The viewers would take the outputs of the image fragments as inputs and display them on separate windows.

Thus, the viewed nodes can have their contents displayed directly on the screen (in viewer windows). Since the screen uses a raster imaging model, the viewer nodes must always have a raster appearance (a bitmap) as their input.

This restriction leads to Figure 11, which shows a viewer connected so as to display a "line segment appearance", i.e., an image fragment containing some Euclidean line segments. Since viewers can only display raster appearances, the connection between the viewer and line segment appearance must be mediated by a "scan conversion" (the circular node 20i) which produces a raster appearance corresponding to the line segment appearance.

The ACG of Figure 11 is particularly interesting since it contains a transform node. Transforms are functional image-transforms which are used in the graph whenever it is desired to alter or combine various image fragments (or transformed image fragments). In the example of Figure 11, the operator desired to view line segments. However, the line segments are drawn using a Euclidean figural model, and in the present example, the viewers can display only appearances drawn using a raster figural model. Accordingly, transform 20i, which transforms Euclidean images into raster images (a scan converter) is placed between the viewer node and the source image node to produce an ACG which views the desired appearance.

Transforms can be used to perform many functions. They are used to transform images within a single figural model, to convert images from one figural model to another, to extract features from images for use in other images, and to combine images from the same or different figural models. Transforms serve both as the tools used to cut images apart and the glue used to put them back together. The scan converter 20i of Figure 11 is conceptually (and computationally) a simple transform. However, the present invention permits arbitrarily complicated transforms which embed arbitrarily complex computations to be incorporated. The only requirement on transforms is that they be functional in the programmatic sense, i.e., their output must be determined solely by their inputs without respect to any internal state. The extensibility of the present invention comes from the ability to add new transforms.

To illustrate further the function of some simple transforms, reference is made to Figures 12A and 12B. Figure 12A illustrates a line segment appearance which is scan converted and then viewed. However, this time the scan converter is parameterized, i.e., it takes additional inputs which determine the size of the scan converted lines and the background pattern that they are drawn with. Thus, the scan conversion node has three children: the line segments to be converted, the background pattern to use when drawing the rasterized lines, and a point (i.e., an x,y pair) which determines the size of the rasterized lines.

The ACG of Figure 12B illustrates a network where one transform's output is the input of another.

In this network, a polygon is extracted (registered) from an arrangement of line segments and then is used to mask a background pattern. The ACG of Fig. 12B is similar to a portion of the Fig. 8 ACG in that a polygon is found using image sources A and B, the found polygon is scan-converted, and then input to viewer node C. The transform 91 is the "polygon finder". It takes a line segment input (from the source labeled A) and a point input (from the point source B), searches outwards from the point in the line segments to find an enclosing polygon, and then outputs that polygon (in this case, the polygon found is a quadrilateral). Transform 95 is a "mask" (or "clipper"). It takes polygon input (from transform 91) and a raster appearance image (from the source labeled F) and produces as its raster appearance output the result of clipping the raster appearance to the interior of the polygon. The output is then displayed by viewer node C.

Any number of different types of image sources can be provided. Some examples include raster appearances, line segment appearances, background patterns, points (whose x- and y-components are always positive), and displacements (points whose components can be negative). Once an image fragment is created - by selecting an image and (optionally) editing, the image fragments can be combined and altered any number of ways, depending on the types of transducers available.

Now, with reference to Figure 13 (which is an enlarged view of the Figure 8 ACG), the ACG illustrated therein and its constructed appearance will be described. The graph will be considered section by section to see how it builds the final appearance. At the lower-left of the ACG is a polygon-producing cluster, similar to the one shown in Figure 12B, consisting of line segment appearance A, point B and a polygon finder transform T10. The output of this cluster is used in a variety of places, so to make it clear what the found polygon looks like, a simple scan-conversion transform T11 is inserted to render the polygon onto viewer C (see Fig. 8). The editor windows 20 and 22 are illustrated to the left of the graph view window 10 in Figure 8. Editor windows for segments A (window 20) and point B (window 22) as well as viewer C are illustrated. Above and to the right of the polygon producing cluster (see Fig. 13), its output is used both by a parameterized scan-conversion T12 and by a mask T13, similar to the one shown in the examples of Figures 12A and 12B. The outputs of these two transforms are then superimposed one on the other (by a superposition transform T14) to produce a combined output which is both shown in Viewer F and used again as a piece of the final, constructed appearance in Viewer G.

The outputs of the parameterized scan-conversion T12 and the mask T13 are also used separately as the inputs to translation transforms T15, T16, T17 which each use a displacement input to determine how far to translate their source appearance. Finally, the output of these translations are then recombined with an underlying background and the superposition discussed via superposition transform T18 above to produce the complete appearance of viewer G.

As can be seen from the above-described example, the present invention, through the use of an ACG-based interface, provides numerous advantages over previous graphics editing systems. It is easy to use multiple image sources without their interfering with one another. Image fragments can be drawn from a variety of figural models. Transformations between models are explicit and consistent. Image components can be reused several times in the same image, both directly and after transformation, without copying or confusion as to which is the original. Transformed images are no different in status from original image fragments - both can be used in exactly the same way. Any collection of component images used in the construction of another image can be viewed separately in order to further clarify their contribution to the image as a whole.

There is nothing particularly interesting about ACGs by themselves - they are simply expression graphs such as those found in the abstract syntax trees provided by standard parsers. However, the use of ACGs as a tool (user interface) for constructing and editing images overcomes the problems, discussed above, which persist in existing graphics editors. ACGs are directed, acyclic graphs (DAGs) whose nodes are called transducers. There are three basic types of transducers: sources, transforms, and sinks. Sources are image generators: they take no children. Transforms are image functions: they take the outputs of their children as inputs and produce a single output which results from combining and transforming these inputs. Sinks are image consumers: they can appear only as roots of the ACG.

ACG's typically consist of one or more source nodes, which could appear as children of one or more transform nodes, which could be children of one or more sink nodes. The sink node in this example, i.e., the viewer, consumes an image formed in the ACG and automatically displays the image in a window whenever the ACG is evaluated. Thus, viewer nodes are used to specify exactly which of the source or transform nodes in the graph are desired to be viewed. Since the ACGs are DAGs, not trees, an operator can both view the output of a node and use that node as the input of other (transform) nodes.

The typical construction process for an ACG interleaves the construction and connection of nodes with the editing of source nodes and the evaluation of the graph to produce output images. Feedback from the evaluation process is then used as a basis for structural alterations in the graph and further edit-

ing of the source images. Thus, the process of ACG construction and revision is analogous to that of program construction and revision, which is natural since ACG's can be thought of as programs (or expressions) which produce images.

The edges in an ACG run from named, typed output ports on one node to named, typed input ports on another. Only one edge can run between any two given ports. The type of an ACG node determines the number, names and types of its ports. The type of an input port determines the allowable types of output ports it can be connected to. Additionally, input port type determines whether the port is unary (can accept only one incoming edge) or n-ary (can accept any number of incoming edges ordered in some sequence). Output ports can always be connected to any number of outgoing edges; their type is used only to determine which input ports those edges can be connected to. A well-formed ACG is a DAG all of whose ports have at least one edge connected to them and all of whose edges obey the type restrictions of their ports.

In the present embodiment, the ACG framework is a single mechanism which provides both a user interface that makes complex appearance construction precise and convenient and an implementation model that makes figural model extensibility and integration precise and convenient.

The user interface presents a simple "erector set" view of images. Images are readily seen to be constructed from small, reusable image fragments and operators. This scheme has three important features. First, it allows for the precise specification, modification, and viewing of the structure of complex images. Second, it allows underlying commonalities across domains (figural models) to be easily captured and reused. Third, it provides a simple and convenient interface to extensibility.

Current WYSIWYG graphics editors have developed a number of features aimed at allowing the user to navigate through and manipulate components of complex images. Grouping and ungrouping of image elements, locking figures in place on the surface, the use of multiple layers (or planes) on which figures can be distributed, the ability to hide figures (make them temporarily invisible) or to magnify them. Mechanisms such as these have been developed, in part, because the WYSIWYG discipline makes it hard to see, and therefore to manipulate, the structure of complex images. By providing a view in which the structure of an image is readily apparent, the present invention makes it easy for the user to see and refer to the elements of the image.

Even though previous techniques such as grouping, locking and layering are of some use in manipulating images with many elements, they provide no help to the user in getting at the relations among elements. For example, none of these techniques provides any help if an image has shared elements (as in Figure 4, where the polygon serves both as a clipping region and as a border). For another example, consider creating a "bitmap polygon" by scan-converting a vector polygon (as can be done in some commercial graphics editors, such as SuperPaint®). Once the scan-conversion is through, the relationship between the figures is lost: changes made to the vector polygon have no effect on the bitmap. Contrast this behavior with that of Figure 7.

The display of ACGs is not biased toward particular figural models: it works equally well for raster graphics as for vector graphics -- and for text, video, or 3D graphics. Nor is it biased toward the details of particular models -- to the presence or absence of particular operators. It therefore provides a convenient interface to the editor's extensible graphics engine. Thus, the same basic mechanism (ACGs) that allows for extensibility suggests a simple and powerful interface to that extensibility.

For purposes of illustration, windows and menus for an exemplary editor will now be provided.

An operator provides commands, for example, by using screen displayed menus and a mouse. The graph view window is displayed at all times. Additionally, a prompt window can be displayed to provide the operator with feedback. The first menu, the Ops menu - which is always enabled - contains commands which apply to the system operation as a whole. The Graph Ops menu - which is enabled only when the graph view window is active - contains commands which apply to the graph view window. The exact content of the menus and the use of the graph view window are briefly described below.

The Prompt Window

The prompt window is used by all system commands whenever they wish to prompt the operator, inform an operator of their progress, or warn an operator of an error. The prompt window cannot be selected since it is intended for output only.

Viewer and Editor Windows

Two other types of windows can be created and destroyed by user request. Viewer windows show the images input to viewer nodes. The contents of these windows cannot be edited directly, rather an operator must edit and re-execute the ACG to cause the viewer windows to be filled with the desired contents. Viewer windows are created automatically whenever the ACG is executed. One viewer window is created for each viewer node, and preferably, the title of the window (which matches the title of the node) is associated with that window. These windows can be closed, however, they will reappear the next time the ACG is evaluated. To cause a viewer window to be deleted

permanently, the viewer node is merely deleted from the ACG.

Editor windows are created when an operator issues a command to edit a source node. Each source node has its own type of editor. Editor windows are titled to match their sources (as with viewers). Editor windows are closed when the editing functions are completed. Some editor window types can have special menus which get added to the menu bar when an editor window of that type is created. These menus will be enabled only when an editor window of that type is active, and will go away if all the editor windows of that type are closed. The specific editors can be implemented using techniques in the prior art. Since all the sources can be imported, commercial, existing editors can be used. All that is required is to provide a means for translating the user input commands to commands which can be understood by the software. This can be accomplished in a conventional manner.

The Ops Menu

The Ops menu permits an operator to select, for example, the following (and other) commands which effect the state of the editor as a whole:

"Edit Graph" makes the graph view window the active window. The editing operations available in this window are described below.

"Execute Graph" evaluates the ACG in the graph view window, and updates all viewer windows appropriately. Progress of the command is shown, for example, in the Prompt Window, as are reports of any errors encountered while executing the graph.

"Save ACG" prompts for a file in a directory (or in a disc file) and saves the current ACG to that file for later reuse.

The Graph View Window

The graph view window is used to construct and edit the ACG. In this window, commands can be issued to create, remove, connect, and disconnect transducer nodes. These commands can be issued, for example, using mouse clicks and drags.

Node Types

Figure 9 is a list of the icons used in the graph view window for the various types of nodes. The input/output protocols for these nodes will now be described.

Sink Nodes

As mentioned above, one type of sink node is the type viewer 20a. Viewer nodes can have only one child, and their function is to display the output of that child, which, in the present examples, must be a raster image. Viewer nodes have no outputs, and so cannot be the children of other nodes. Whenever an ACG is executed, a viewer window is created for each viewer node in the ACG. This viewer window displays the input of the viewer node. Both the node icon and the viewer window are labeled with, for example, a letter from A to Z so it is clear which viewer node's inputs is being displayed in each view window. Other types of sink nodes could, for example, send images to files for storage, or to a printer (or plotter) for printing.

Transforms

Each transform type is characterized by its input (their type and cardinality) and how its output is generated from its input. A first type of transform is the superposition transform 20g. This transform takes any number of children. The output of each child must be a raster image. The output of the superposition (a raster image) is obtained by superposing (layering) the outputs of the children in order, so that the inputs of later children obscure (are layered on top of) the inputs from earlier children. The order of the children is determined by the order in which the superposition is connected to its inputs, however, other ordering techniques can be used. In this example, the child which is connected first gets layered first and is covered by the child connected second, which is covered by the child connected third, and so on.

A second type of transform is the scan conversion transform 20i. This transform takes a single child, which must be a line-segment vector image, and produces as its output a raster image obtained by scan-converting the line segments with a width of one pixel and a black pen pattern.

Another type of scan converter is a parameterized scan converter. This transform takes three children. One must be a line-segment vector image, another must be a point source, and the third must be a background source. The order of the children does not matter. The output of the transform is a raster image obtained by scan converting the line segments with a pen whose width is the x-value of the point source, whose height is the y-value of the point source, and whose pattern is the background pattern. The default values for the x and y components can be, for example, 1, while the default background pattern can be black.

These two types of scan converters could be offered as separate choices, or as a single choice (with the default values for background and pen-size operating as the non-parameterized scan converter).

A third type of transform is the polygon-finder 20j. This transform takes two children, a line segment image and a point source, in any order. Its output is a line segment image whose line segments form a polygon. The output polygon is the smallest polygon which can

be registered in the line segment image that contains the point whose coordinates are given in the point source. If there is no such polygon, then evaluating the polygon- finder node will produce an error.

A fourth type of transform is a mask 20k. This transform takes two children, a raster image and a line segment image (whose segments should form a polygon), in any order. The output of the mask is a raster image consisting of the input raster clipped to the interior of the polygon.

A fifth type of transform is a translation transform 20h. This transform takes two children, a displacement source and an image (either a raster image or a line-segment image), in either order. Its output is the input image translated by the displacement.

Source Nodes

Source nodes, as mentioned above, produce an output but take no inputs. Thus, source nodes have no children. Each source node, when first created, for example, can be given a default output value (contents) based on its type. The source node can then be edited in order to change its contents. Similarly, source images can be imported from external files (e.g. discs) to produce the content of a source node.

The display icon for each source node in an ACG is labeled with, for example, a letter from "A" to "Z", so that source nodes can be distinguished easily. If an operator asks to edit a source node, an editor window is created and labeled with that node's letter, so it is always clear which editor window goes with which node.

Different source types use different types of editor windows. The editor window for each source type is described with the source type.

A first source type is a point 20f. A point is an ordered pair (x,y) of non-negative, integer coordinates. The default value for a newly created point node can be (0,0). Points are edited using a dialogue window that has a text item for each of the coordinates, an OK button, and a cancel button. The window labeled Point F in Figure 3 is a point source editor window. The x and y coordinate values can be edited by entering new values and buttoning OK.

A second type of source is a displacement node 20e. A displacement is an ordered pair (x,y) of integers (either negative or positive). The default value for a newly created displacement node can be, for example, (0,0). Displacements are edited exactly as points are.

A third type of source is a background source 20b. A background is a raster image of indefinite size filled with, for example, an 8 pixel square repeating image called a pattern. Backgrounds are taken to extend infinitely in all directions, i.e., they will completely fill any size pen (when used for scan conversion) or mask. The default pattern for a newly-created background node can be, for example, all white. Backgrounds are edited in a special dialogue window which permits an operator to choose from a variety of predefined patterns. The window labelled Background C in Figure 3 illustrates a window displaying some illustrative patterns. The selected background pattern is indicated by enclosing it in a black square. The background pattern can be changed by buttoning a different pattern and then buttoning OK. Buttoning cancel will move the black square back to the original background pattern.

A fourth type of source node is a raster appearance 20c. A raster appearance is a previously prepared raster image, i.e., a bitmap. Raster appearances are edited using a special dialogue window which allows an operator to choose from a variety of prepared raster images. The window labeled Raster Constant D in Figure 3 is an example of the special dialogue window for raster appearances. The selected raster image is highlighted by, for example, placing a box around it. The images are described rather than shown, although a display which shows the images could also be used. An operator can change the raster appearance's bitmap by buttoning a different name and buttoning OK.

A fifth type of source is a line segment appearance 20d. A line segment appearance is a set of Euclidean line segments, i.e., a set of infinitely thin line segments whose end points have real coordinates. Editing a line segment appearance will now be described.

As described earlier, line segment appearances cannot be viewed directly, rather they must always be scan converted and the results viewed. Accordingly, the operating system has an editor window for a line segment appearance node display of the result of applying a scan-conversion transform to the node segments. The window labeled Viewer C in Figure 8 illustrates such a conversion. The editing operations in line segment appearance editor windows are invoked from a menu that is enabled whenever such an editor is active. When an operator creates a line segment appearance node in the graph view window, that node is automatically connected to a scan conversion node and a viewer node, and the editing window for that viewer node is then displayed so that the operator can edit the line segment appearance. The line segment appearance can be edited any number of ways which are conventional in the art. New segments can be created, existing segments can be deleted, and any segments can be moved.

Implementation

The present invention is implemented in a manner which allows for flexible, precise, simple integration of new figural operations and figural models (graphical domains) into existing operations and

models. Programmers can add new operations for use with existing models (e.g., affine transformation over vector art), new operations between existing figural models (e.g., raster-to-vector recognition operators, vector-to-raster scan-conversion operations), and entirely new figural models (e.g., text, 3D, video) both with their own operations and with existing-model inter-conversion operations.

The extensibility of implementation, just like the precision of the user interface, is rooted directly in ACGs. The key to this is the observation that ACGs are basically data flow diagrams. The basis of the implementation of the present graphics editor is the direct representation of ACGs in an object-oriented graph representation language. For example, Common Lisp Object System (CLOS) graph representation language can be used. Nodes in the graph are objects which keep track of their connections to neighbors and which respond to a standard set of graph traversal messages. The graphic functionality of transducers is implemented by making their objects inherit functionality not only from the node class (i.e., the type of input/output-source, transforms, sink) but also from a graphics operator class (i.e., perform specific function based on input) which responds to standard operation messages such as reading inputs and providing outputs. Transducer node types (all node types are illustrated in Fig. 9) correspond to subclasses of the generic operator class which implement functionality appropriate to the node type.

As a user uses the graph-construction interface to build an ACG, the editor is in fact building this ACG in the underlying object model. When the operator asks that the ACG be executed, a graph traversal algorithm is run which, as it walks the graph from the bottom up (from leaf nodes to root nodes), invokes the graphics operations associated with each source, operator, and sink. This has the effect of constructing and displaying the appropriate appearance. In other words, the present invention executes ACGs by propagating graphics information along the data flow arcs which connect the ACG nodes.

This very simple, direct implementation model has two great advantages. First, as mentioned early, it allows for the easy integration both of new operators and of new figural models. Second, it allows traditional compiler optimization technology, already adapted for use with abstract syntax graphs, to be employed directly so aS to reduce both the time and space requirements for graph execution.

If a programmer wished to integrate some new figural functionality, expressed perhaps as a library of routines and some associated data structures, two aspects must be addressed. Appropriate access must be provided to the operation library, and appropriate typing information must be provided to the ACG construction interface. However, since in the present invention, both operation invocation and graph construction are handled via the same mechanism, one achieves both simply by writing some object-oriented "glue" interfaces to the library. (And, possibly if one is greatly concerned with efficiency, some optimization advice to the graph walker.)

The implementation of the present invention is best understood as being broken into four interconnected modules represented by a block diagram in Fig. 14:

1) ACG support module 1410. This module provides a representation of the ACG graphs, specifies the input/output protocols for the transducers embedded in ACGs, and implements execution of ACGs.

2) Graphics support module 1420. This module implements the figural models which are supported by the graphics editor.

3) Transducer module 1430. This module provides the implementations of the particular transducers used in the graphics editor.

4) User interface module 1440. This module provides for the editing of ACGs and image fragments, and for the display of generated images.

ACG support

The ACG module supports two related views of ACGs: one - crucial to the user interface, as directed acyclic graphs whose nodes are transducers; another - crucial to image generation - as expressions in a functional imaging language whose operators are transducers. Thus, an ACG can be manipulated as a directed, acyclic graph (DAG) - creating and destroying transducer nodes, adding and removing edges between nodes - and then asking the editor to produce one or more appearances from that graph by evaluating the expression it represents - running each transducer in the graph in the proper sequence over the proper inputs.

The ACG support module represents ACGs directly as DAGs. The expression view is supported by a graph traversal (evaluation) algorithm which recursively descends an ACG from the roots to the fringes, each node in the ACG is evaluated by applying its transformation to the inputs obtained by evaluating its children. As an operator manipulates the graph view in the user interface, that module constructs an appropriate ACG using the tools provided by this module. Then, when an operator asks to see the image produced by the ACG, the user-interface module uses the evaluation traversal to produce an output image at each of the roots of the prepared graph, and then displays those images.

The representation of ACG DAGs makes use of a special metaclass (i.e., "graphic-operator "). Instances with metaclass "graphic-operator" act as nodes of ACGs. The nodes can have an arbitrary number of named ports each of which can contain

children, and they keep back pointers to all their parents. All the standard graph operations are defined over nodes by defining methods over the special class "operator" which is automatically made a super class (i.e., raster, vector, etc.) of any instance with metaclass "graphic-operator". These operations include both queries (How many children does this node have? What are its port names? etc.) and manipulations (add this child to this port of this node, etc.).

The evaluation traversal for ACGs works by invoking an evaluation method (i.e., "perform-draw-actions") on the roots of the ACG. This method is specialized on each class of transducer so as to perform the operations appropriate to that class. Any transducer which takes inputs (has children in the graph) will invoke "perform- draw-actions" recursively on its inputs. This evaluation proceeds post-order from the roots to the fringes in an ACG. Thus, a parent will not be evaluated until all of its children are evaluated.

This evaluation algorithm invokes "perform-draw- actions" on a node at least once for each parent that the node has. Since ACGs are DAGs, rather than trees, this may lead to nodes being evaluated more than once during a single evaluation pass. Since transducers are required not to have side effects, however, this repetitive evaluation cannot affect the output produced by the evaluation process, only the efficiency of evaluation. Since efficiency must be of concern in any real-time editor, ACG evaluation can be optimized.

One class of optimizations is suggested by the observation that while the present invention evaluates the graphic-language expression represented by an ACG, it would be possible instead to compile such an expression into a more efficient form, and then execute that. If the additional cost of compilation were sufficiently small, then the combined cost of the compilation and successive execution would be smaller than that of the original evaluation. Three natural optimizations suggested by this approach are:

1) Repeated-expression removal. As observed above, repeated evaluations of the same subgraph of an ACG must produce identical results. In fact, repeated evaluations of isomorphic subgraphs must also produce the same results (where two fringe nodes are considered equivalent if they contain the same image fragments). After analyzing the expression graph and picking out isomorphic subgraphs, evaluation of the subgraphs could be done just once and their output cached for reuse as necessary.

2) Unreachable-expression removal. Not all the output of a node (or subgraph) is always used by the parent of that node. Pre-analysis of the ACG can reveal such situations, allowing the removal of unused sub- expressions and a consequent savings of execution time.

3) Idempotent-expression removal. Sometimes the output of a transducer is identical to its input. Pre- analysis of the ACG can reveal such situations, allowing the removal of such transducers and a subsequent saving of execution time.

Implementation of these three optimizations can be achieved by augmenting ACGs so that they are attribute- graphs (such as those discussed in the standard compiler- theory literature) and then doing specialized attribute propagation as a precursor to the evaluation traversal.

One more optimization-incremental reevaluation- could be added to the ACG module in order to improve the response to changes made in an existing ACG. Since an ACG is often reevaluated by a user after making changes in small pieces of that ACG, and since unchanged subgraphs will always produce the same output on subsequent evaluations, reevaluations could often be speeded up by not re-evaluating unchanged portions of the ACG.

Graphics support

In the present embodiment, in order to support the Euclidean models, the software must provide rudimentary support for the finding of segment-point distances and segment-segment intersections. The raster model, and conversions from the Euclidean to the raster models can be supported, for example, by QuickDraw®, a proprietary set of routines made available for general use on the Macintosh by Apple Computer.

Transducers

Transducers have two types of functionality. They are the nodes in ACGs, thus they have a graph functionality which involves keeping track of their children and their parents. They are also appearance transformers, thus they have a graphics functionality which involves manipulating the appearances of their children so as to produce an output appearance.

As explained above, the graph functionality of transform instances is implemented by making them have a metaclass operator-class. This gives each transform instance one slot for each of its named ports: these slots keep track of the instance's children, and graph operations on the instance work by making changes in the slots.

Also as explained above, the graphic functionality of transform instances is implemented by defining special evaluation methods over them. That is, for each class of node, a method definition for "perform-draw-actions" is defined which performs a specified function on a specific class (or classes) of input to create a specific type of output.

A distinction is made between transducers that

implement fringe nodes of ACGs (which are image fragments) and those that implement internal nodes (transforms). Since the graphic functionality of transducers is determined by method definitions, and since method definitions are class-based, the class of a transducer is the primary determiner of its behavior. What differentiates the behavior of instances of the same transducer class is the local state of that instance, namely its slot contents. We have already seen that each transducer stores its children in its slots, thus, the behavior of internal nodes (transforms) can be made dependent on their children. In fact, the implementation of internal transducers (transforms) are consistently dependent only on their class and their children. Thus, they are functionally defined.

Fringe nodes (image sources), on the other hand, have no children, and so instances of the same class cannot be differentiated thereby. Instead, each fringe transducer is given an internal state which can be thought of as an image fragment in one of the figural models, and the graphic functionality of the transducer is determined by that image fragment. This is why the user interface provides editors for fringe nodes, but not for internal nodes. Internal nodes have no state to edit other than their children, while fringe nodes do.

The important thing about the transducer frame work is that it makes building extensions very modular. The graphic functionality desired can simply be couched as methods attached to a particular class. The input and output of this functionality are then automatically attended to by the ACG module.

User Interface

The user interface breaks into three parts: ACG editing, image-fragment (or source) editing, and viewers. Each part is extremely straightforward.

The ACG editing mechanism works by interpreting signals input by a user (for example, by using mouse clicks and drags) as step-by-step instruction for building a DAG. (Other means for interpreting operator input commands, of course, can be provided.) It then follows these instructions two-fold: it builds a special interface DAG whose nodes and edges keep information about the graph window's display, and it also uses the ACG module to build the appropriate ACG. This strategy of keeping a separate interface DAG (rather than annotating the DAG with display information) was adopted to increase the implementation's modularity. Major changes can be made in both the user interface DAG representation and the ACG DAG representation without any interference between the two.

The source editing mechanism consists of unrelated, dedicated editors, one for each type of source (line segments, prepared bit maps, backgrounds, points, and displacements). Four of the five editors can be, for example, Macintosh-style dialogues which allow a user to choose from a variety of pre-known possibilities.

The line-segment editor, however, recursively uses the transducer mechanism of the present invention. The basic problem in building an editor for line segment appearances which are defined using a Euclidean model, is that the editing interface -the display screen - uses a raster model. This problem must be faced by all editors whose input and output device models differ from the figural models of the appearances they edit. Thus, there is no way to display the appearance being edited directly , nor is there any way to use mouse pointing to specify directly end points for segments. Both the input and output of segments must make use of some raster representation.

The line-segment editor of the present invention handles this problem by recursively using the transducer network. When a user wishes to edit a line segment image source, a small ACG consisting of that source node, a single scan-conversion transducer, and a viewer as described above, is created. The viewer of this ACG is used as the viewer of the segment editor, thus the user sees the same representation of the segments as they would if they used a scan-converter transducer in the ACG. For purposes of referring to segments, the coordinate system of the line segment domain and the coordinate system of the raster domain are kept synchronized, so that mouse clicks appearing in the raster domain can be interpreted directly as mouse clicks in the vector domain.

Figure 15 is a flowchart illustrating the steps performed to create and display a complex image using the present invention. Images are created by constructing the ACG on the display screen in step 1510, editing the source node contents in step 1520, and executing the ACG in step 1530. After ACG execution, the constructed and any intermediate images indicated in the ACG can be displayed, for example, on the display screen in step 1540. Should any changes in the constructed image be desired after execution of the ACG, the ACG can be edited in step 1550, the source node contents can be edited, and then the ACG can be re-executed . While an entire ACG can be constructed before editing any of the source nodes, an operator will preferably edit the source nodes as they are created. Additionally, the ACG and source nodes can be modified and remodified prior to execution of the ACG.

Figure 16 illustrates the steps performed to construct an ACG on a display screen. As described earlier, the graph view window is activated in step 1511, by, for example, the user providing the appropriate signal (called up from the Graph Ops menu). The user then inputs a signal (e.g., by manipulating a mouse in the graph view window and inputting mouse clicks

where nodes are desired in the window) to indicate where a node is desired. The operator is then prompted to specify the node type in step 1513. If the node type is a source node, the operator may choose to edit the contents of the source node's image fragment at this time. The nodes are interconnected in step 1515. Interconnecting nodes requires that the operator input signals which specify the two nodes to be interconnected in step 1516. This can be done, for example, by positioning the cursor over one node, depressing a key on the keyboard, depressing the button on the mouse, and dragging the cursor to the desired node. When the cursor is released (completing the drag) by releasing the keyboard and cursor buttons, the two nodes are connected. This is displayed on the screen as a line interconnecting the two nodes. Additionally, the underlying data structure takes the output of the child node and uses it as an input to the parent node. Prompts can be provided to indicate that the specified child is not of the proper type input for the parent. These prompts are issued based upon evaluations of the previously described protocols for each node type. It is understood that the specific user manipulations for creating nodes, specifying node types, and interconnecting nodes can vary based upon the software used to implement the present invention. Additionally, the operator can connect newly created nodes to existing nodes as they are created, and does not have to wait until all nodes in the ACG are formed before connecting any of the nodes.

Figure 17 is a flow diagram of the process for executing the image formation process defined by an ACG. In step 1531, an operator inputs an "execute ACG" command. At this time, the structure walker can determine whether the ACG is well formed in step 1532. A well formed ACG is a DAG all of whose ports have at least one edge connected to them and all of whose edges obey the type restrictions of their ports. If the ACG is not well formed, the user is prompted in step 1533 so that they can edit the ACG prior to re-execution. If the ACG is well formed, operation proceeds to step 1534 where the nodes of the ACG are evaluated from the fringes (leafs) to the roots, as discussed above.

Figures 18A and 18B illustrate the processes for adding new nodes and deleting nodes, respectively, from an existing ACG. Referring to Figure 18A, when an operator provides a control signal for adding a new node in step 1551, the operator is prompted to specify the node type in step 1552. If the node type is a source node, the operator may specify the contents of the source node at this time. Once a new node is defined, the operator then interconnects that node to one or more existing nodes in the ACG in step 1554. This is performed by specifying the nodes for interconnection to the newly defined node in step 1555. It is understood that the process of interconnecting two nodes can be performed at any time in order to edit an ACG. That is, an already existing node can be connected to another existing node by performing step 1555.

When it is desired to delete nodes, the operator provides a control signal in step 1557 to delete the specified node. This can be done, for example by positioning a cursor over a node and inputting an appropriate signal (e.g., hold down a keyboard key and mouse click over the node) indicating that this node should be deleted. Once the node is deleted, the interconnections which existed between that node and other nodes must be deleted in step 1558. This can be performed automatically when nodes are deleted. Additionally, interconnections (edges) between existing nodes can be deleted by providing the appropriate signals to delete a specified edge.

Claims

1. An electronic image generator, comprising:

   means for generating at least one output image capable of being displayed, the output image being generated as a result of performing no, one or some transformation operations on one or more source images;

   means for constructing and displaying a graphical flow diagram of an image formation process used to generate the output image, the graphical flow diagram including a plurality of interconnected nodes which represent the source images, any transformation operations, and the output image; and

   means for controlling the graphical display constructing means to form and modify said graphical flow diagram, wherein the output image generation is responsive to the said graphical flow diagram so as to generate the or each output image.

2. The electronic image generator of claim 1, wherein the graphical flow diagram is acyclic.

3. The electronic image generator of claim 1 or 2, wherein the graphical flow diagram corresponds to an image formation process operating on plural source images in which each source image may be subjected to transformation operations, in which the output of transformation operations may be themselves subjected to transformation operations, and in which both source images and the outputs of transformation operations may be selectively used as the output images.

4. The electronic image generator of claim 3, wherein output images result directly or indirectly from all applications of transformation operations.

5. The electronic image generator of any preceding claim, wherein the control means is responsive to operator input commands.

6. The electronic image generator of any preceeding claim, wherein the control means causes the means for generating a constructed image to display the constructed image concurrently with the displayed graphical flow diagram.

7. The electronic image generator of any preceding claim, wherein the graphical flow diagram has nodes which are generated and displayed in a vertically-ordered manner, such that the nodes representing transformation operations appear above the nodes representing source images or other transformation operations to which they are being applied, and the nodes representing output images appear above the nodes representing source images or the transformation operations to which they are connected.

8. The electronic image generator of any preceeding claim, wherein the control means permits the selection of source images and transformation operations from a library.

9. The electronic image generator of claim 8, further comprising means for defining and entering additional source images and transformation operations into the library.

10. The electronic image generator of any preceding claim, further comprising means for translating image data defining the source images or outputs of transformation operations into different graphical domains to permit the image data to be processed by transformation operations operating in different graphical domains.

11. The electronic image generator of claim 10, wherein the means for translating permits translation of image data into a graphical domain suitable for display.

12. The electronic image generator of claim 10 or 11, wherein the different graphical domains include at least two-dimensional rasters and two-dimensional vectors.

13. The electronic image generator of claim 12, wherein the transformation operations selectively operate in a single graphical domain or across different graphical domains so that each transformation operation has input protocols and output protocols, and wherein the control means controls the means for constructing and displaying a graphical flow diagram so that the graphical flow diagram is constructed according to the input and output protocols.

14. The electronic image generator of any preceding claim, wherein the means for generating one or more output images can use source images and the results of performing transformation operations on source images as inputs to a plurality of further transformation operations, the means for constructing a graphical flow diagram displaying such plural input use by attaching the node associated with the source images and the results to the further transformation operations.

15. The electronic image generator of any preceding claim, further comprising a display screen, wherein the output images and the graphical flow diagram of an image formation sequence used to produce the output images are generated on the display screen.

16. A method of generating a graphical representation of an image formation process, comprising:

selecting at least one image source used as a component of one or more final output images;

selecting no, one or more transformation operations to be performed on the image sources to produce, at least indirectly, the final output images; and

generating a graphical flow diagram including display icons representing all selected image sources, all selected image transformation operations, all selected final output images, and interconnections between the image sources, the image transformation operations, and the final output images which represent the image formation sequence which is performed to produce the final output images.

17. A method of constructing a graphics image represented by an output image display icon from one or more image sources represented by image source display icons and no, one or more transformation operations represented by transformation operation display icons, each transformation operation capable of being performed on at least one of the image sources, the method comprising:

selecting at least one of the image sources as a component of a final output image;

optionally selecting a transformation operation to be performed on the image source to produce, at least indirectly, the final output image;

generating a graphical flow diagram of an image formation sequence required to produce the final output image by generating on a display screen the display icons associated with the se-

lected image sources, the selected transformation operations, the final output image, and interconnections between the image source display icons, the transformation operation display icons, and the output image display icon which represent the image formation sequence which is performed to produce the final output image, and

generating the final output image by executing the selected transformation operations on the selected image sources according to the graphical flow diagram.

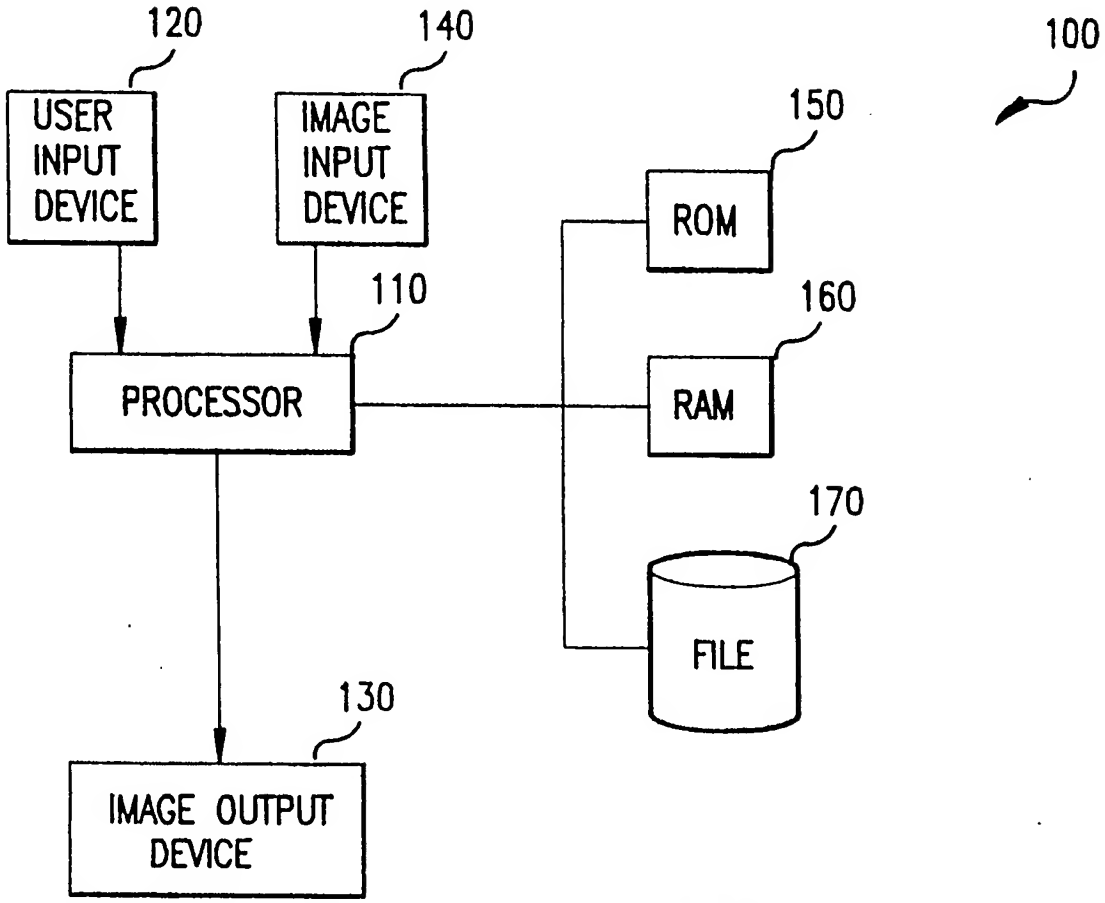

120
140
100
USER INPUT DEVICE
IMAGE INPUT DEVICE
150
ROM
110
160
PROCESSOR
RAM
170
FILE
130
IMAGE OUTPUT DEVICE

FIG.1

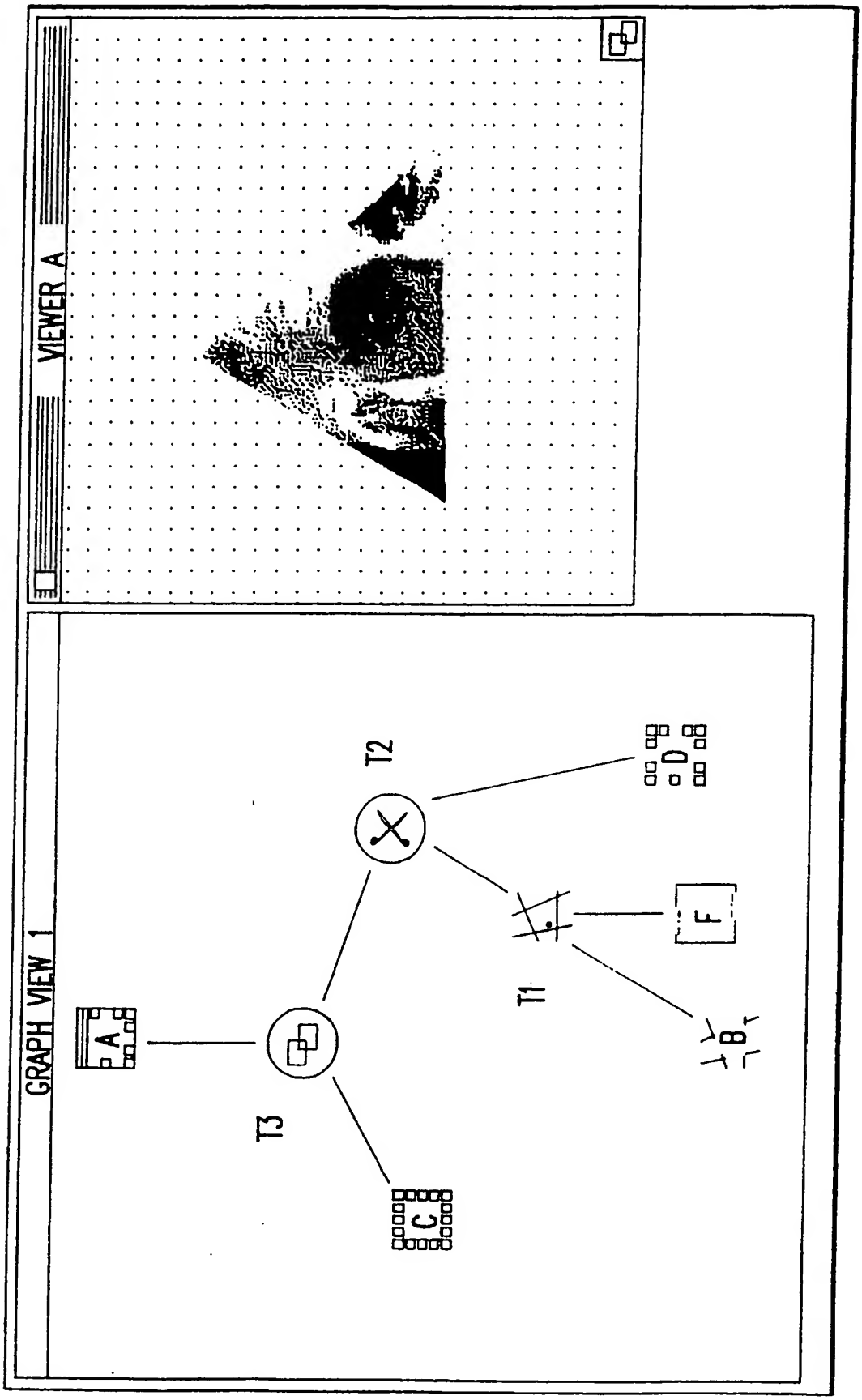
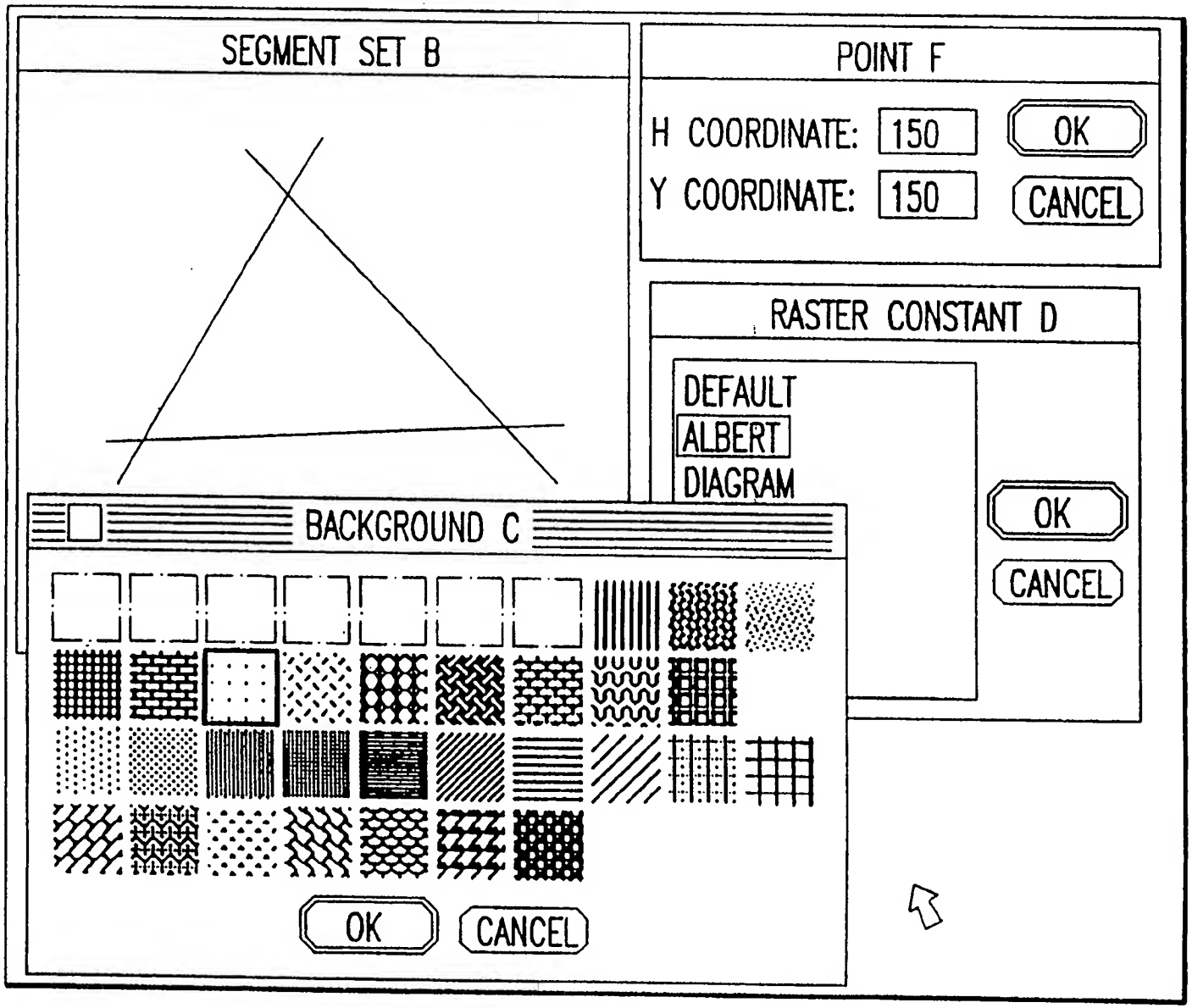


FIG.2


SEGMENT SET B
POINT F
H COORDINATE:
150
OK
Y COORDINATE:
150
CANCEL
RASTER CONSTANT D
DEFAULT
ALBERT
DIAGRAM
OK
CANCEL
BACKGROUND C
OK
CANCEL

20

FIG.3

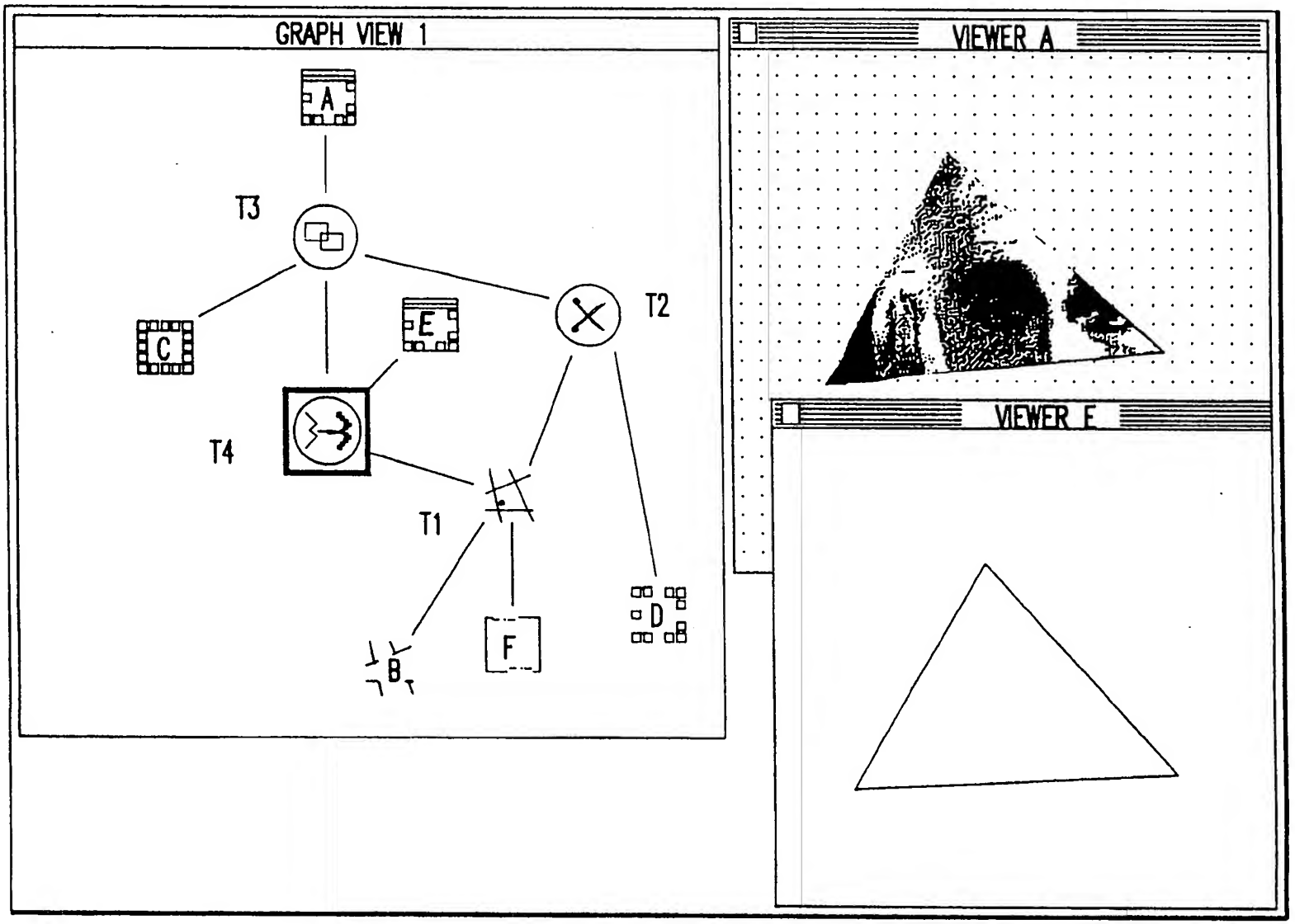


FIG.4

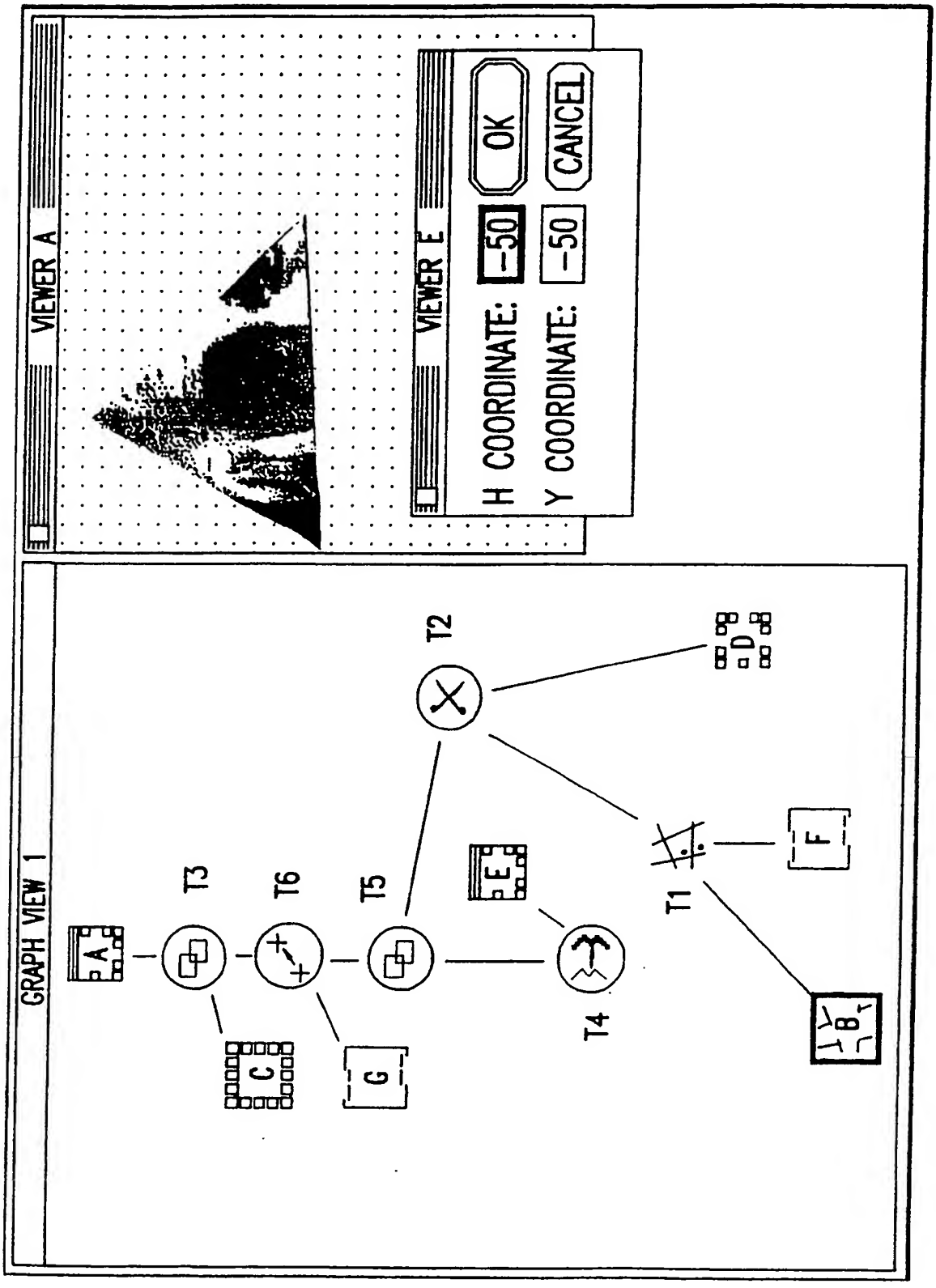

VIEWER A
VIEWER E
H COORDINATE:
-50
Y COORDINATE:
-50
OK
CANCEL
GRAPH VIEW 1
A
B
C
D
E
F
G
T1
T2
T3
T4
T5
T6

FIG.5

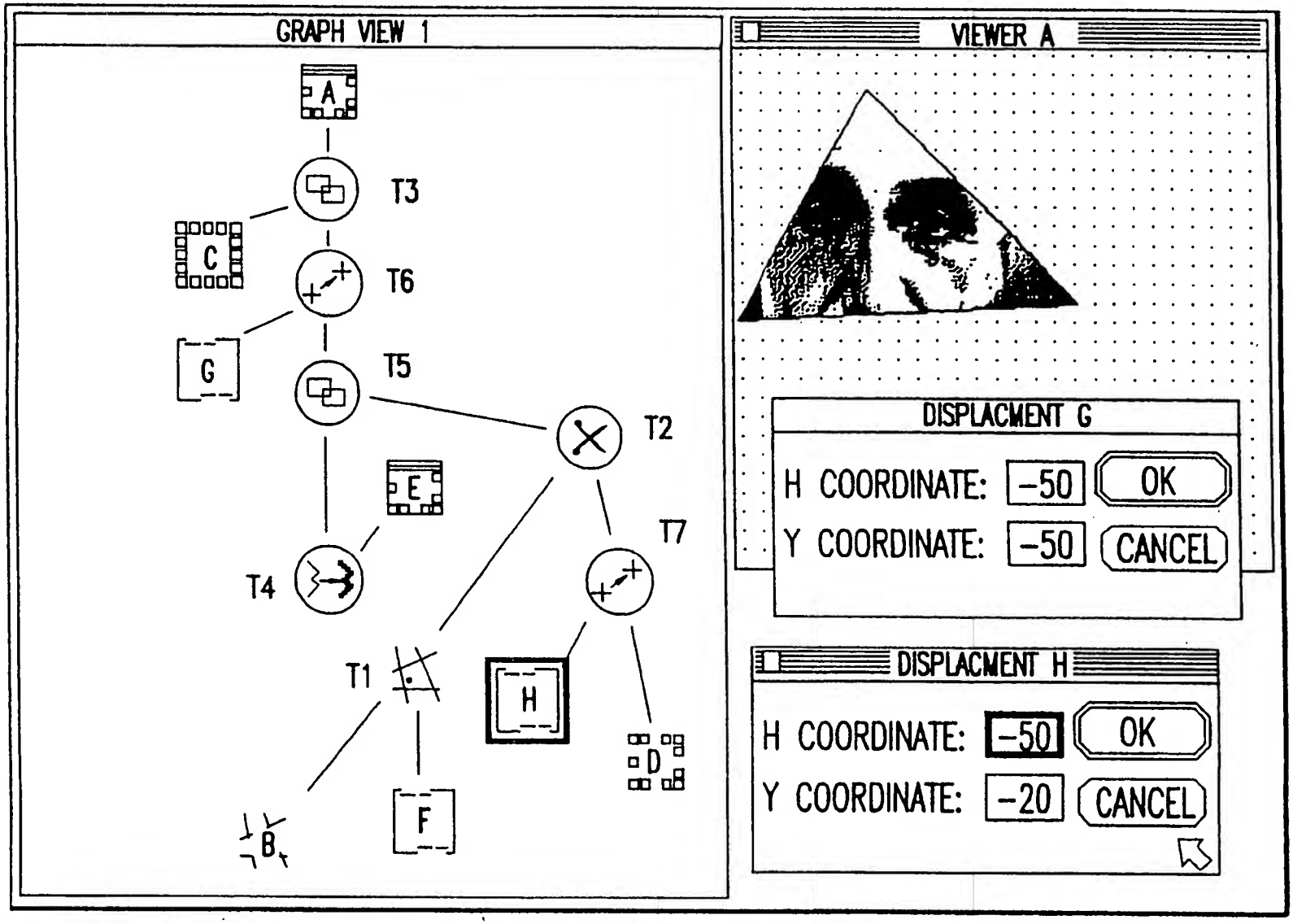

GRAPH VIEW 1
A
T3
C
T6
G
T5
T2
E
T7
T4
T1
H
D
F
B
VIEWER A
DISPLACMENT G
H COORDINATE: -50
OK
Y COORDINATE: -50
CANCEL
DISPLACMENT H
H COORDINATE: -50
OK
Y COORDINATE: -20
CANCEL

FIG.6

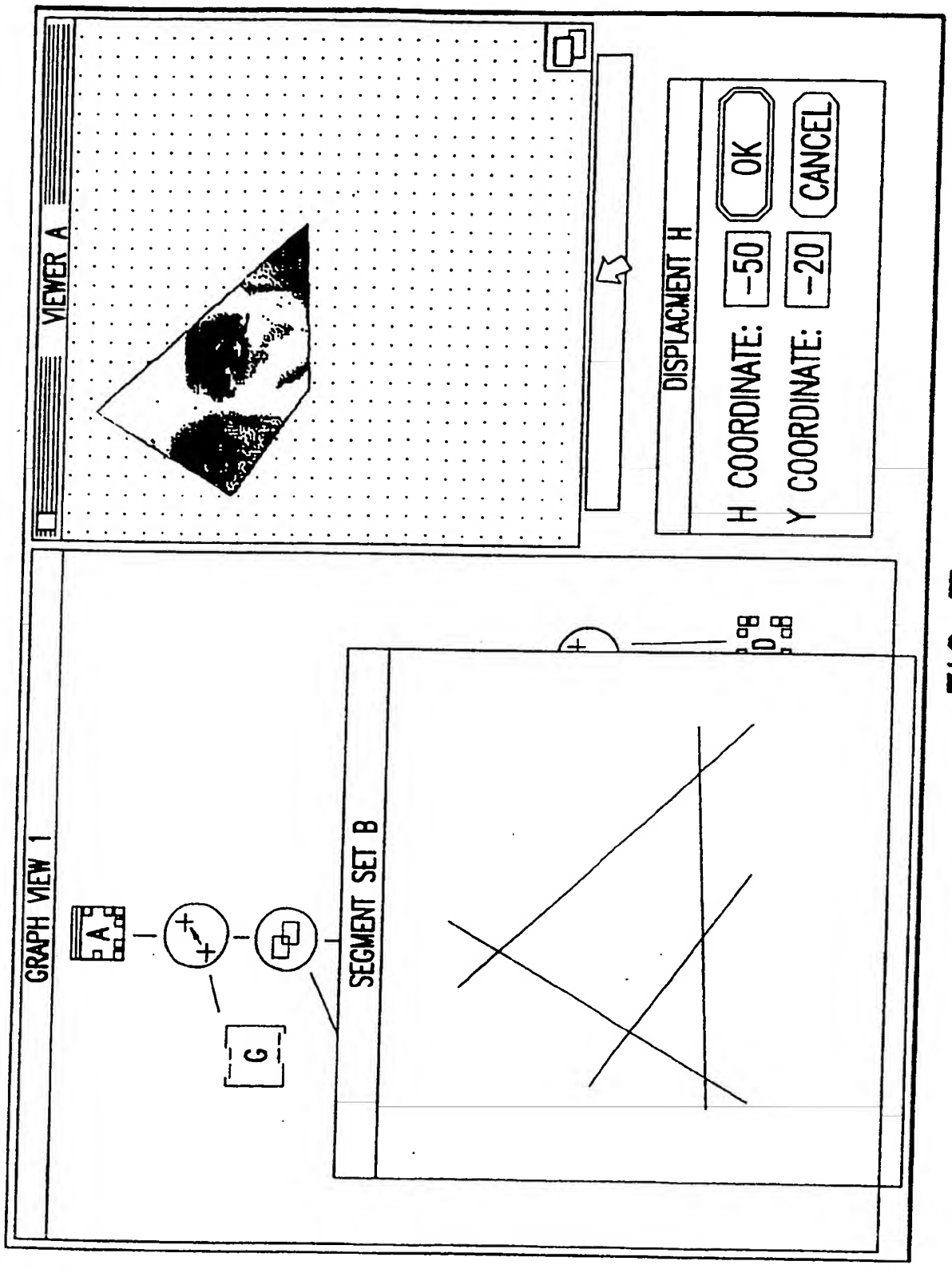

VIEWER A
DISPLACMENT H
H COORDINATE:
-50
OK
Y COORDINATE:
-20
CANCEL
GRAPH VIEW 1
A
G
SEGMENT SET B
D

FIG. 7

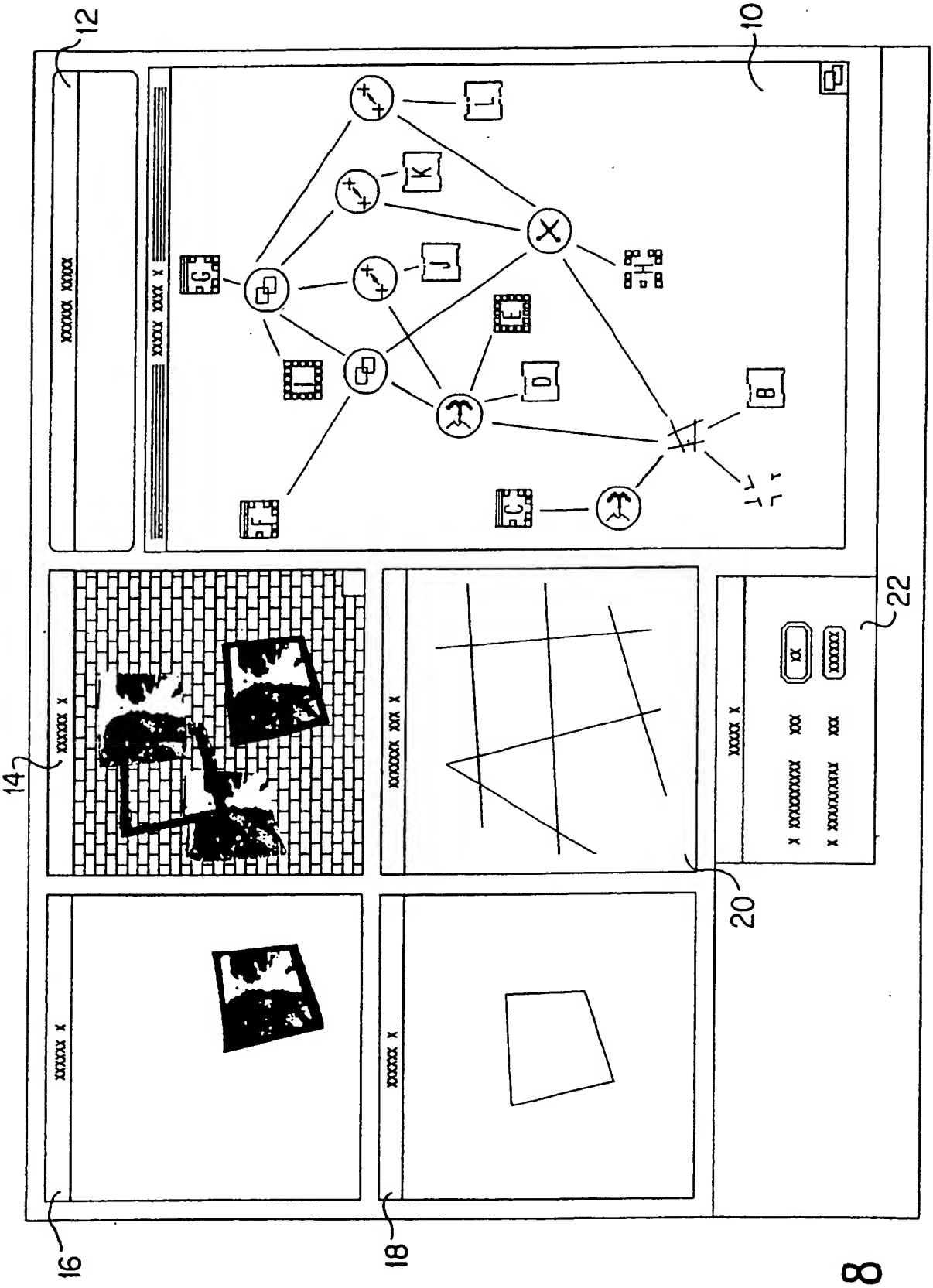


FIG.8

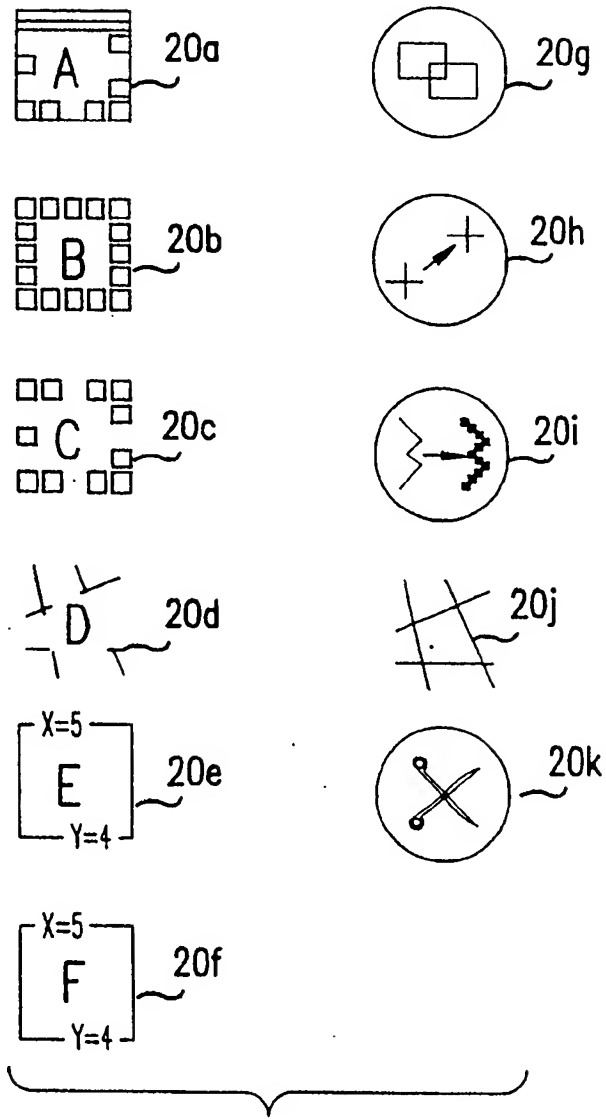


FIG.9

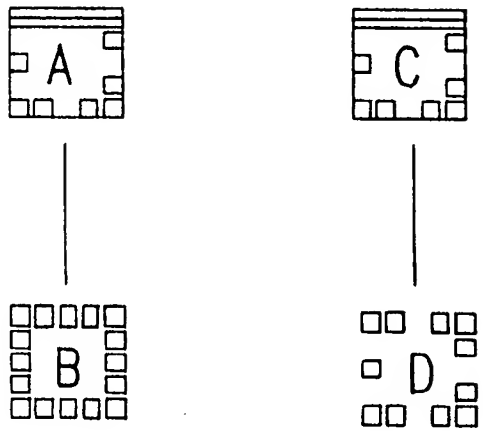


FIG.10A

FIG.10B

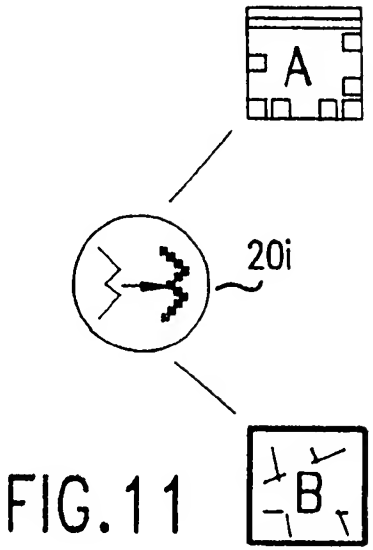


FIG.11

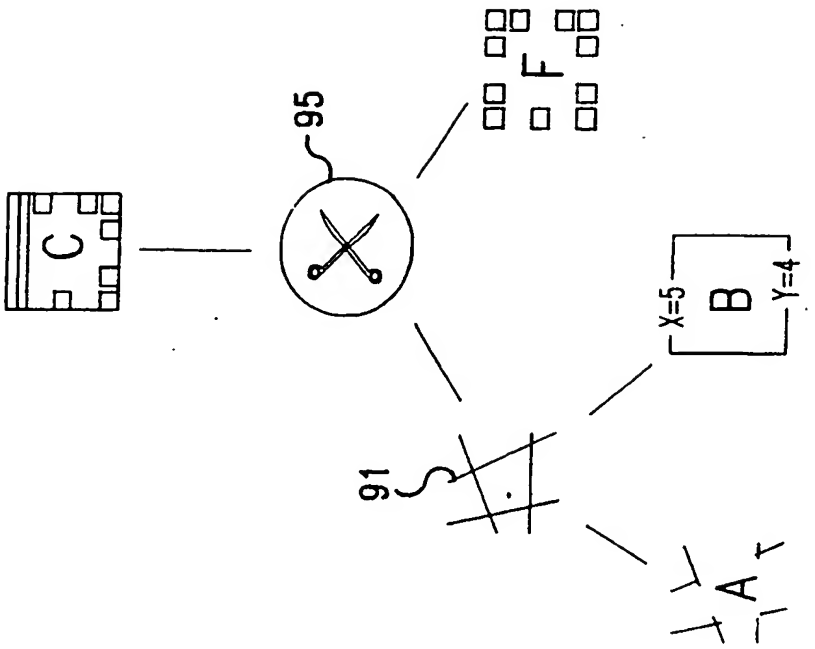


FIG.12B

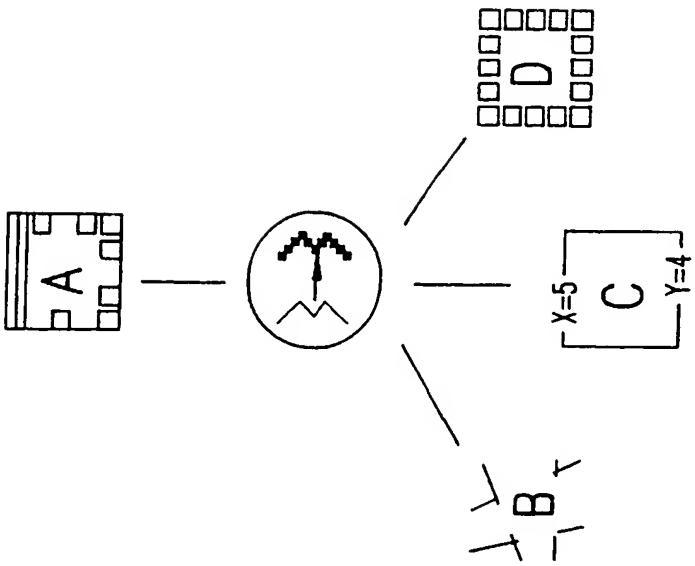


FIG.12A

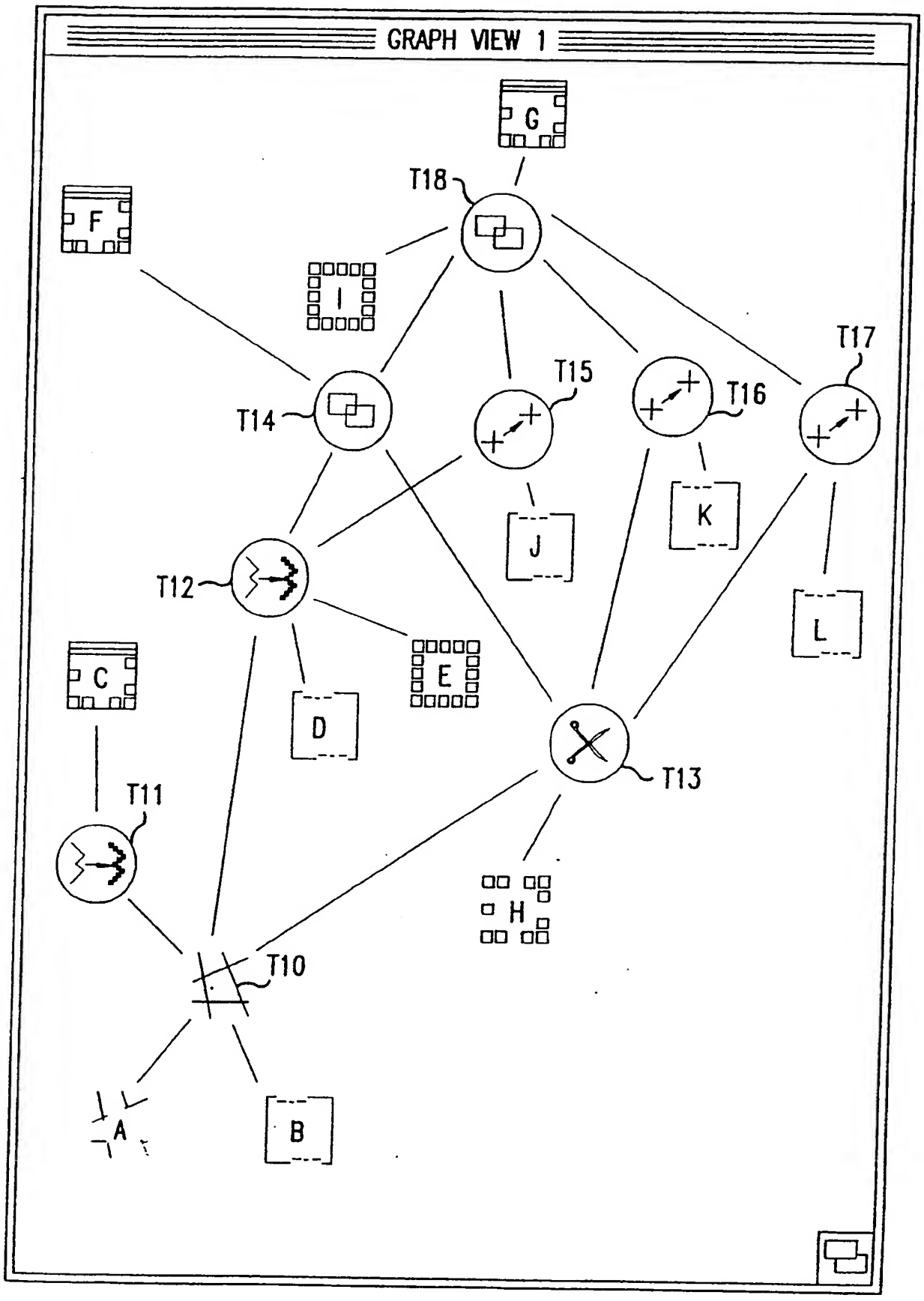


FIG.13

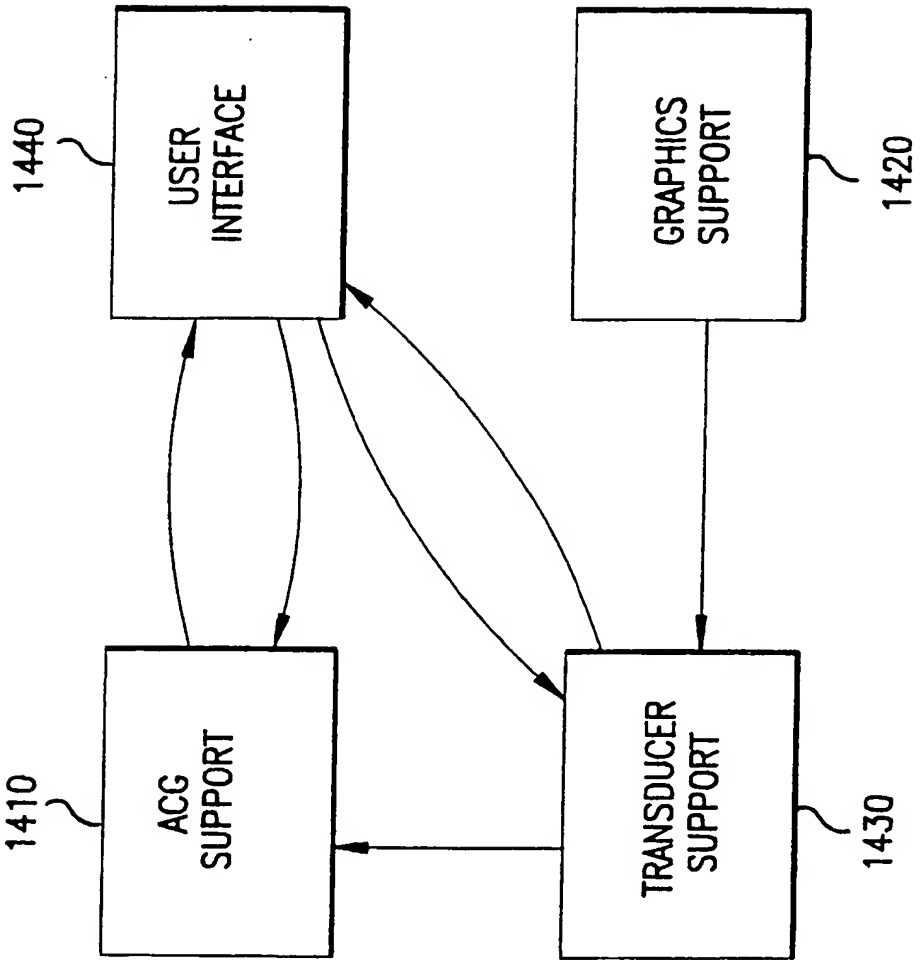

1440
USER INTERFACE
GRAPHICS SUPPORT
1420
1410
ACG SUPPORT
TRANSDUCER SUPPORT
1430

FIG.14

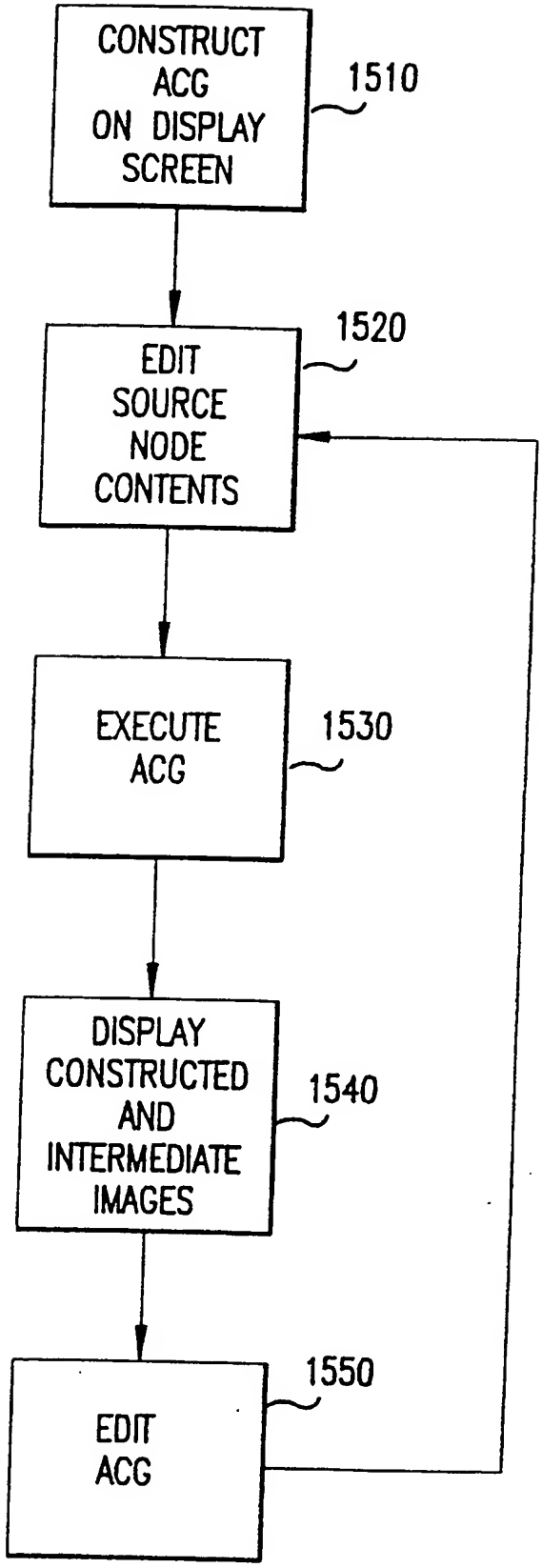

CONSTRUCT ACG ON DISPLAY SCREEN
1510
EDIT SOURCE NODE CONTENTS
1520
EXECUTE ACG
1530
DISPLAY CONSTRUCTED AND INTERMEDIATE IMAGES
1540
EDIT ACG
1550

FIG.15

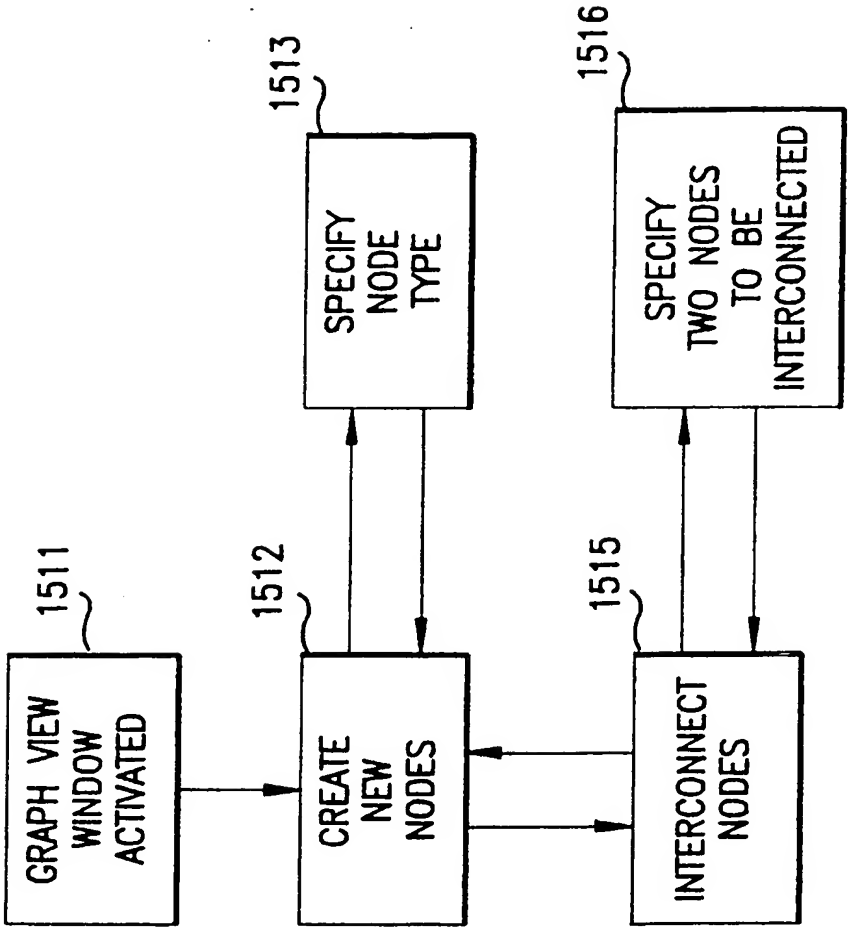

1511
GRAPH VIEW WINDOW ACTIVATED
1512
CREATE NEW NODES
1513
SPECIFY NODE TYPE
1515
INTERCONNECT NODES
1516
SPECIFY TWO NODES TO BE INTERCONNECTED

FIG.16

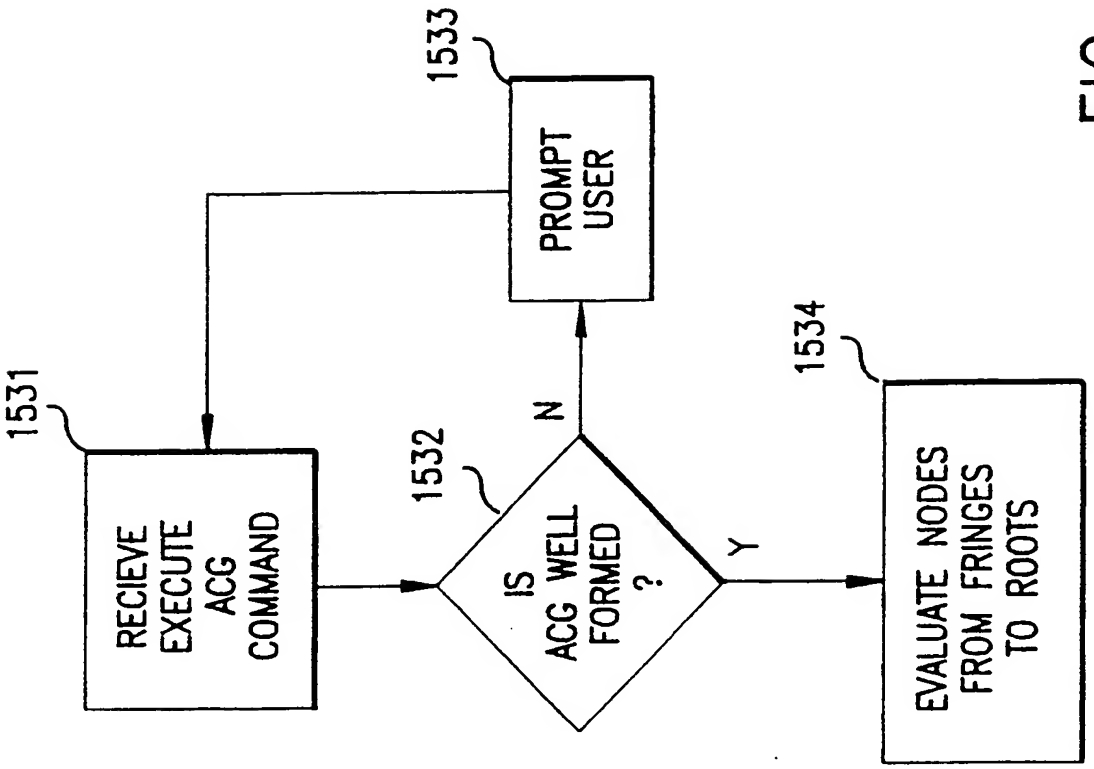

1531
RECIEVE EXECUTE ACG COMMAND
1532
IS ACG WELL FORMED ?
N
1533
PROMPT USER
Y
1534
EVALUATE NODES FROM FRINGES TO ROOTS

FIG.17

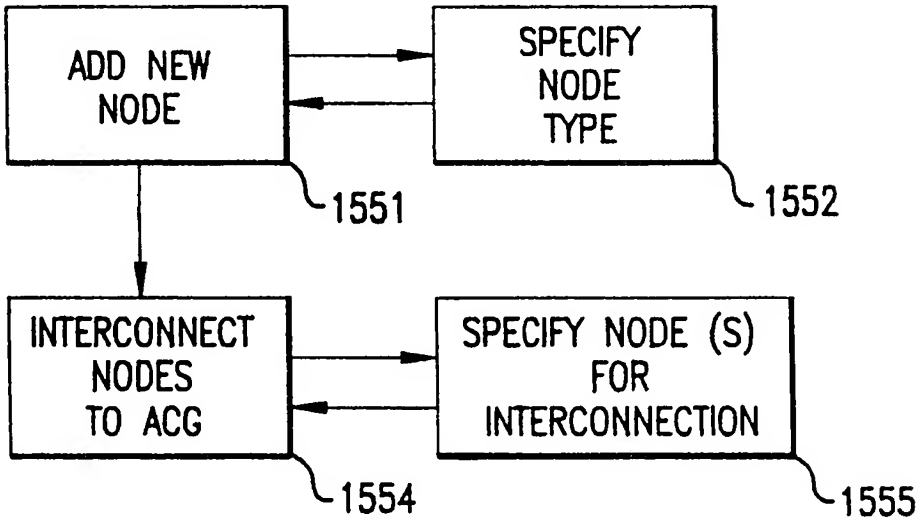


FIG.18A

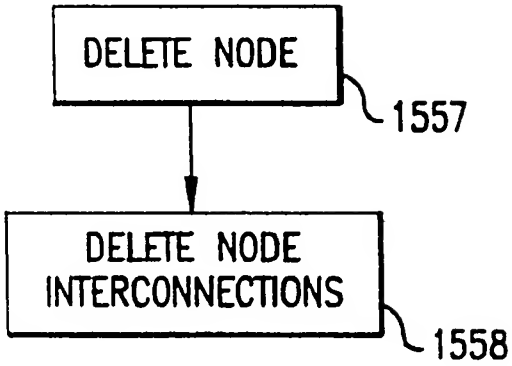


FIG.18B